Prolog

programming for artificial intelligence

Third edition

Prolog programming for artificial intelligence

Third edition

IVAN BRATKO

Faculty of Computer and Information Science,
Ljubljana University
and
J. Stefan Institute

Addison-Wesley

An imprint of Pearson Education

Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney
Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoneduc.com

First edition 1986
Second edition 1990
Third edition 2001

ISBN 0201-40375-7

British Library Cataloguing-in-Publication Data
A catalogue record for this book can be obtained from the British Library

Library of Congress Cataloging-in-Publication Data
Bratko, Ivan 1946-
Prolog programming for artificial intelligence / Ivan Bratko. -- 3rd ed.
p. cm. -- (International computer science series)
Includes bibliographical references and index.
ISBN 0-201-40375-7
1. Artifical intelligence--Data processing. 2. Prolog (Computer program language)
I. Title. II. Series.
Q336.B74 2001
006.3'0285'5133--dc21 00-026973

10 9 8 7 6 5 4 3 2
05 04 03 02 01

Typeset by 43 in 9/12.5pt Stone Serif
Printed in Great Britain by Henry Ling Ltd, at the Dorset Press, Dorchester, Dorset

I dedicate the third edition of this book
to my mother, the kindest person I know
and to my father, who, during world war II
escaped from a concentration camp by
digging an underground tunnel, which he
described in his novel, *The Telescope*

Contents

From Patrick Winston's Foreword to the Second Edition

I can never forget my excitement when I saw my first Prolog-style program in action. It was part of Terry Winograd's famous Shrdlu system, whose blocks-world problem solver arranged for a simulated robot arm to move blocks around a screen, solving intricate problems in response to human-specified goals.

Winograd's blocks-world problem solver was written in Microplanner, a language which we now recognize as a sort of Prolog. Nevertheless, in spite of the defects of Microplanner, the blocks-world problem solver was organized explicitly around goals, because a Prolog-style language encourages programmers to think in terms of goals. The goal-oriented procedures for grasping, clearing, getting rid of, moving, and ungrasping made it possible for a clear, transparent, concise program to seem amazingly intelligent.

Winograd's blocks-world problem solver permanently changed the way I think about programs. I even rewrote the blocks-world problem solver in Lisp for my Lisp textbook because that program unalterably impressed me with the power of the goal-oriented philosophy of programming and the fun of writing goal-oriented programs.

But learning about goal-oriented programming through Lisp programs is like reading Shakespeare in a language other than English. Some of the beauty comes through, but not as powerfully as in the original. Similarly, the best way to learn about goal-oriented programming is to read and write goal-oriented programs in Prolog, for goal-oriented programming is what Prolog is all about.

In broader terms, the evolution of computer languages is an evolution away from low-level languages, in which the programmer specifies how something is to be done, toward high-level languages, in which the programmer specifies simply *what* is to be done. With the development of Fortran, for example, programmers were no longer forced to speak to the computer in the procrustian low-level language of addresses and registers. Instead, Fortran programmers could speak in their own language, or nearly so, using a notation that made only moderate concessions to the one-dimensional, 80-column world.

Fortran and nearly all other languages are still how-type languages, however. In my view, modern Lisp is the champion of these languages, for Lisp in its Common Lisp form is enormously expressive, but how to do something is still what the Lisp programmer is allowed to be expressive about. Prolog, on the other hand, is a language that clearly breaks away from the how-type languages, encouraging the

programmer to describe situations and problems, not the detailed means by which the problems are to be solved.

Consequently, an introduction to Prolog is important for all students of Computer Science, for there is no better way to see what the notion of what-type programming is all about.

In particular, the chapters of this book clearly illustrate the difference between how-type and what-type thinking. In the first chapter, for example, the difference is illustrated through problems dealing with family relations. The Prolog programmer straightforwardly describes the grandfather concept in explicit, natural terms: a grandfather is a father of a parent. Here is the Prolog notation:

```
grandfather( X, Z) :- father( X, Y), parent( Y, Z).
```

Once Prolog knows what a grandfather is, it is easy to ask a question: who are Patrick's grandfathers, for example. Here again is the Prolog notation, along with a typical answer:

```
?- grandfather( X, patrick).

X = james;
X = carl
```

It is Prolog's job to figure out how to solve the problem by combing through a database of known father and parent relations. The programmer specifies only what is known and what question is to be solved. The programmer is more concerned with knowledge and less concerned with algorithms that exploit the knowledge.

Given that it is important to learn Prolog, the next question is how. I believe that learning a programming language is like learning a natural language in many ways. For example, a reference manual is helpful in learning a programming language, just as a dictionary is helpful in learning a natural language. But no one learns a natural language with only a dictionary, for the words are only part of what must be learned. The student of a natural language must learn the conventions that govern how the words are put legally together, and later, the student should learn the art of those who put the words together with style.

Similarly, no one learns a programming language from only a reference manual, for a reference manual says little or nothing about the way the primitives of the language are put to use by those who use the language well. For this, a textbook is required, and the best textbooks offer copious examples, for good examples are distilled experience, and it is principally through experience that we learn.

In this book, the first example is on the first page, and the remaining pages constitute an example cornucopia, pouring forth Prolog programs written by a passionate Prolog programmer who is dedicated to the Prolog point of view. By carefully studying these examples, the reader acquires not only the mechanics of the language, but also a personal collection of precedents, ready to be taken apart, adapted, and reassembled together into new programs. With this acquisition of

precedent knowledge, the transition from novice to skilled programmer is already under way.

Of course, a beneficial side effect of good programming examples is that they expose a bit of interesting science as well as a lot about programming itself. The science behind the examples in this book is Artificial Intelligence. The reader learns about such problem-solving ideas as problem reduction, forward and backward chaining, 'how' and 'why' questioning, and various search techniques.

In fact, one of the great features of Prolog is that it is simple enough for students in introductory Artificial Intelligence subjects to learn to use immediately. I expect that many instructors will use this book as part of their artificial intelligence subjects so that their students can see abstract ideas immediately reduced to concrete, motivating form.

Among Prolog texts, I expect this book to be particularly popular, not only because of its examples, but also because of a number of other features:

- Careful summaries appear throughout.
- Numerous exercises reinforce all concepts.
- Structure selectors introduce the notion of data abstraction.
- Explicit discussions of programming style and technique occupy an entire chapter.
- There is honest attention to the problems to be faced in Prolog programming, as well as the joys.

Features like this make this a well done, enjoyable, and instructive book.

I keep the first edition of this textbook in my library on the outstanding-textbooks shelf, programming languages section, for as a textbook it exhibited all the strengths that set the outstanding textbooks apart from the others, including clear and direct writing, copious examples, careful summaries, and numerous exercises. And as a programming language textbook, I especially liked its attention to data abstraction, emphasis on programming style, and honest treatment of Prolog's problems as well as Prolog's advantages.

Preface

Prolog

Prolog is a programming language centred around a small set of basic mechanisms, including pattern matching, tree-based data structuring and automatic backtracking. This small set constitutes a surprisingly powerful and flexible programming framework. Prolog is especially well suited for problems that involve objects – in particular, structured objects – and relations between them. For example, it is an easy exercise in Prolog to express spatial relationships between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y, and Y is closer than Z, then X must be closer than Z. Prolog can now reason about the spatial relationships and their consistency with respect to the general rule. Features like this make Prolog a powerful language for artificial intelligence (AI) and non-numerical programming in general. There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code. When the same algorithms were implemented in Prolog, the result was a crystal-clear program easily fitting on one page.

Development of Prolog

Prolog stands for *programming in logic* – an idea that emerged in the early 1970s to use logic as a programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side), Maarten van Emden at Edinburgh (experimental demonstration) and Alain Colmerauer at Marseilles (implementation). David D.H. Warren's efficient implementation at Edinburgh in the mid-1970s greatly contributed to the popularity of Prolog. A more recent development is *constraint logic programming* (CLP), usually implemented as part of a Prolog system. CLP extends Prolog with constraint processing, which has proved in practice to be an exceptionally flexible tool for problems like scheduling and logistic planning. In 1996 the official ISO standard for Prolog was published.

Historical controversies about Prolog

There are some controversial views that historically accompanied Prolog. Prolog fast gained popularity in Europe as a practical programming tool. In Japan, Prolog was placed at the centre of the development of the fifth-generation computers. On the

other hand, in the United States Prolog was generally accepted with some delay, due to several historical factors. One of these was an early American experience with the Microplanner language, also akin to the idea of logic programming, but inefficiently implemented. Some reservations also came in reaction to the early 'orthodox school' of logic programming, which insisted on the use of pure logic that should not be marred by adding practical facilities not related to logic. This led to some widespread misunderstandings about Prolog in the past. For example, some believed that only backward chaining reasoning can be programmed in Prolog. The truth is that Prolog is a general programming language and any algorithm can be programmed in it. The impractical 'orthodox school's' position was modified by Prolog practitioners who adopted a more pragmatic view, benefiting from combining the new, declarative approach with the traditional, procedural one.

Learning Prolog

Since Prolog has its roots in mathematical logic it is often introduced through logic. However, such a mathematically intensive introduction is not very useful if the aim is to teach Prolog as a practical programming tool. Therefore this book is not concerned with the mathematical aspects, but concentrates on the art of making the few basic mechanisms of Prolog solve interesting problems. Whereas conventional languages are procedurally oriented, Prolog introduces the descriptive, or *declarative*, view. This greatly alters the way of thinking about problems and makes learning to program in Prolog an exciting intellectual challenge. Many believe that every student of computer science should learn something about Prolog at some point because Prolog enforces a different problem-solving paradigm complementary to other programming languages.

Contents of the book

Part I of the book introduces the Prolog language and shows how Prolog programs are developed. Techniques to handle important data structures, such as trees and graphs, are also included because of their general importance. In Part II, Prolog is applied to a number of areas of AI, including problem solving and heuristic search, programming with constraints, knowledge representation and expert systems, planning, machine learning, qualitative reasoning, language processing and game playing. AI techniques are introduced and developed in depth towards their implementation in Prolog, resulting in complete programs. These can be used as building blocks for sophisticated applications. The concluding chapter, on meta-programming, shows how Prolog can be used to implement other languages and programming paradigms, including object-oriented programming, pattern-directed programming and writing interpreters for Prolog in Prolog. Throughout, the emphasis is on the clarity of programs; efficiency tricks that rely on implementation-dependent features are avoided.

Differences between the second and third edition

All the material has been revised and updated. There are new chapters on:

- constraint logic programming (CLP);
- inductive logic programming;
- qualitative reasoning.

Other major changes are:

- addition of belief networks (Bayes networks) in the chapter on knowledge representation and expert systems;
- addition of memory-efficient programs for best-first search (IDA*, RBFS) in the chapter on heuristic search;
- major updates in the chapter on machine learning;
- additional techniques for improving program efficiency in the chapter on programming style and technique.

Throughout, more attention is paid to the differences between Prolog implementations with specific references to the Prolog standard when appropriate (see also Appendix A).

Audience for the book

This book is for students of Prolog and AI. It can be used in a Prolog course or in an AI course in which the principles of AI are brought to life through Prolog. The reader is assumed to have a basic general knowledge of computers, but no knowledge of AI is necessary. No particular programming experience is required; in fact, plentiful experience and devotion to conventional procedural programming – for example in C or Pascal – might even be an impediment to the fresh way of thinking Prolog requires.

The book uses standard syntax

Among several Prolog dialects, the Edinburgh syntax, also known as DEC-10 syntax, is the most widespread, and is the basis of the ISO standard for Prolog. It is also used in this book. For compatibility with the various Prolog implementations, this book only uses a relatively small subset of the built-in features that are shared by many Prologs.

How to read the book

In Part I, the natural reading order corresponds to the order in the book. However, the part of Section 2.4 that describes the procedural meaning of Prolog in a more

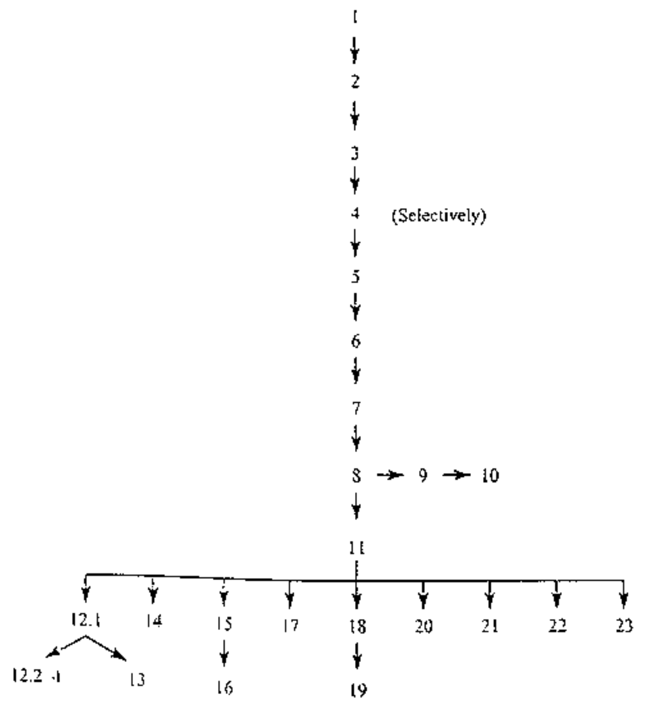


Figure P.1 Precedence constraints among the chapters.

formalized way can be skipped. Chapter 4 presents programming examples that can be read (or skipped) selectively. Chapter 10 on advanced tree representations can be skipped.

Part II allows more flexible reading strategies as most of the chapters are intended to be mutually independent. However, some topics will still naturally be covered before others, such as basic search strategies (Chapter 11). Figure P.1 summarizes the natural precedence constraints among the chapters.

Program code and course materials

Source code for all the programs in the book and relevant course materials are accessible from the companion web site (www.booksites.net/bratko).

Acknowledgements

Donald Michie was responsible for first inducing my interest in Prolog. I am grateful to Lawrence Byrd, Fernando Pereira and David H.D. Warren, once members of

the Prolog development team at Edinburgh, for their programming advice and numerous discussions. The book greatly benefited from comments and suggestions to the previous editions by Andrew McGettrick and Patrick H. Winston. Other people who read parts of the manuscript and contributed significant comments include: Damjan Bojadžiev, Rod Bristow, Peter Clark, Frans Coenen, David C. Dodson, Sašo Džeroski, Bogdan Filipič, Wan Fokkink, Matjaž Gams, Peter G. Greenfield, Marko Grobelnik, Chris Hinde, Igor Kononenko, Matevž Kovačič, Eduardo Morales, Igor Mozetič, Timothy B. Niblett, Dušan Peterc, Uroš Pompe, Robert Rodošek, Agata Saje, Claude Sammut, Cem Say, Ashwin Srinivasan, Dorian Šuc, Peter Tancig, Tanja Urbančič, Mark Wallace, William Walley, Simon Weilguny, Blaž Zupan and Darko Zupanič. Special thanks to Cem Say for testing many programs and his gift of finding hidden errors. Several readers helped by pointing out errors in the previous editions, most notably G. Oulsnam and Iztok Tvrdy. I would also like to thank Karen Mosman, Julie Knight and Karen Sutherland of Pearson Education for their work in the process of making this book. Simon Plumtree and Debra Myson-Etherington provided much support in the previous editions. Most of the artwork was done by Darko Simeršek. Finally, this book would not be possible without the stimulating creativity of the international logic programming community.

The publisher wishes to thank Plenum Publishing Corporation for their permission to reproduce material similar to that in Chapter 10 of *Human and Machine Problem Solving* (1989), K. Gilhooly (ed).

Ivan Bratko
January 2000

part I

The Prolog Language

chapter 1

Introduction to Prolog

This chapter reviews basic mechanisms of Prolog through an example program. Although the treatment is largely informal many important concepts are introduced such as: Prolog clauses, facts, rules and procedures. Prolog's built-in backtracking mechanism and the distinction between declarative and procedural meanings of a program are discussed.

1.1 Defining relations by facts

Prolog is a programming language for symbolic, non-numeric computation. It is specially well suited for solving problems that involve objects and relations between objects. Figure 1.1 shows an example: a family relation. The fact that Tom is a parent of Bob can be written in Prolog as:

```
parent( tom, bob).
```

Here we choose **parent** as the name of a relation; **tom** and **bob** are its arguments. For reasons that will become clear later we write names like **tom** with an initial lower-case letter. The whole family tree of Figure 1.1 is defined by the following Prolog program:

```
parent( pam, bob).
parent( tom, bob).
```

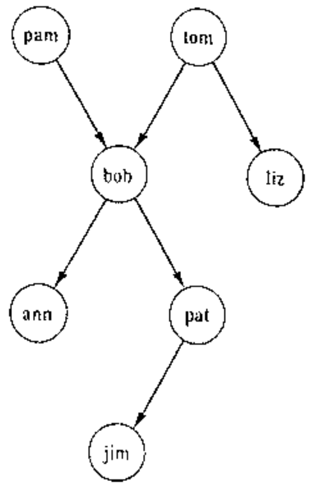


Figure 1.1 A family tree.

```
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

This program consists of six *clauses*. Each of these clauses declares one fact about the parent relation. For example, parent(tom, bob) is a particular *instance* of the parent relation. Such an instance is also called a *relationship*. In general, a relation is defined as the set of all its instances.

When this program has been communicated to the Prolog system, Prolog can be posed some questions about the parent relation. For example: Is Bob a parent of Pat? This question can be communicated to the Prolog system by typing into the terminal:

```
?- parent( bob, pat).
```

Having found this as an asserted fact in the program, Prolog will answer:

```
yes
```

A further query can be:

```
?- parent( liz, pat).
```

Prolog answers:

```
no
```


because the program does not mention anything about Liz being a parent of Pat. It also answers 'no' to the question:

```
?- parent( tom, ben).
```

because the program has not even heard of the name Ben.

More interesting questions can also be asked. For example: Who is Liz's parent?

```
?- parent( X, liz).
```

Prolog's answer will not be just 'yes' or 'no' this time. Prolog will tell us what is the value of X such that the above statement is true. So the answer is:

```
X = tom
```

The question Who are Bob's children? can be communicated to Prolog as:

```
?- parent( bob, X).
```

This time there is more than just one possible answer. Prolog first answers with one solution:

```
X = ann
```

We may now request another solution (by typing a semicolon), and Prolog will find:

```
X = pat
```

If we request more solutions again, Prolog will answer 'no' because all the solutions have been exhausted.

Our program can be asked an even broader question: Who is a parent of whom? Another formulation of this question is:

Find X and Y such that X is a parent of Y.

This is expressed in Prolog by:

```
?- parent( X, Y).
```

Prolog now finds all the parent-child pairs one after another. The solutions will be displayed one at a time as long as we tell Prolog we want more solutions, until all the solutions have been found. The answers are output as:

```
X = pam
Y = bob;

X = tom
Y = bob;

X = tom
Y = liz;

...
```

We can stop the stream of solutions by typing a return instead of a semicolon.

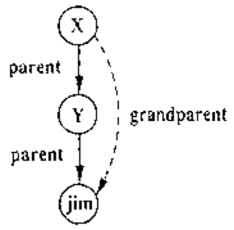


Figure 1.2 The grandparent relation expressed as a composition of two parent relations.

Our example program can be asked still more complicated questions like: Who is a grandparent of Jim? As our program does not directly know the grandparent relation this query has to be broken down into two steps, as illustrated by Figure 1.2.

(1) Who is a parent of Jim? Assume that this is some Y.

(2) Who is a parent of Y? Assume that this is some X.

Such a composed query is written in Prolog as a sequence of two simple ones:

```
?- parent( Y, jim), parent( X, Y).
```

The answer will be:

```
X = bob
Y = pat
```

Our composed query can be read: Find such X and Y that satisfy the following two requirements:

```
parent( Y, jim)   and   parent( X, Y)
```

If we change the order of the two requirements the logical meaning remains the same:

```
parent( X, Y)   and   parent( Y, jim)
```

We can indeed do this in our Prolog program, and the query:

```
?- parent( X, Y), parent( Y, jim).
```

will produce the same result.

In a similar way we can ask: Who are Tom's grandchildren?

```
?- parent( tom, X), parent( X, Y).
```

Prolog's answers are:

```
X = bob
Y = ann;
```



```
X = bob
Y = pat
```

Yet another question could be: Do Ann and Pat have a common parent? This can be expressed again in two steps:

(1) Who is a parent, X, of Ann?

(2) Is (this same) X a parent of Pat?

The corresponding question to Prolog is then:

```
?- parent( X, ann), parent( X, pat).
```

The answer is:

```
X = bob
```

Our example program has helped to illustrate some important points:

- It is easy in Prolog to define a relation, such as the parent relation, by stating the n-tuples of objects that satisfy the relation.
- The user can easily query the Prolog system about relations defined in the program.
- A Prolog program consists of *clauses*. Each clause terminates with a full stop.
- The arguments of relations can (among other things) be: concrete objects, or constants (such as tom and ann), or general objects such as X and Y. Objects of the first kind in our program are called *atoms*. Objects of the second kind are called *variables*.
- Questions to the system consist of one or more *goals*. A sequence of goals, such as:

  ```
  parent( X, ann), parent( X, pat)
  ```

  means the conjunction of the goals:

  X is a parent of Ann, and
  X is a parent of Pat.

  The word 'goals' is used because Prolog accepts questions as goals that are to be satisfied.
- An answer to a question can be either positive or negative, depending on whether the corresponding goal can be satisfied or not. In the case of a positive answer we say that the corresponding goal was *satisfiable* and that the goal *succeeded*. Otherwise the goal was *unsatisfiable* and it *failed*.
- If several answers satisfy the question then Prolog will find as many of them as desired by the user.

Exercises

1.1 Assuming the parent relation as defined in this section (see Figure 1.1), what will be Prolog's answers to the following questions?

(a) ?- parent(jim, X).

(b) ?- parent(X, jim).

(c) ?- parent(pam, X), parent(X, pat).

(d) ?- parent(pam, X), parent(X, Y), parent(Y, jim).

1.2 Formulate in Prolog the following questions about the parent relation:

(a) Who is Pat's parent?

(b) Does Liz have a child?

(c) Who is Pat's grandparent?

1.2 Defining relations by rules

Our example program can be easily extended in many interesting ways. Let us first add the information on the sex of the people that occur in the parent relation. This can be done by simply adding the following facts to our program:

```
female( pam).
male( tom).
male( bob).
female( liz).
female( pat).
female( ann).
male( jim).
```

The relations introduced here are male and female. These relations are unary (or one-place) relations. A binary relation like parent defines a relation between *pairs* of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects. The first unary clause above can be read: Pam is a female. We could convey the same information declared in the two unary relations with one binary relation, sex, instead. An alternative piece of program would then be:

```
sex( pam, feminine).
sex( tom, masculine).
sex( bob, masculine).
...
```

As our next extension to the program let us introduce the offspring relation as the inverse of the parent relation. We could define offspring in a similar way as the

parent relation; that is, by simply providing a list of simple facts about the offspring relation, each fact mentioning one pair of people such that one is an offspring of the other. For example:

```
offspring( liz, tom).
```

However, the offspring relation can be defined much more elegantly by making use of the fact that it is the inverse of parent, and that parent has already been defined. This alternative way can be based on the following logical statement:

For all X and Y,
  Y is an offspring of X if
    X is a parent of Y.

This formulation is already close to the formalism of Prolog. The corresponding Prolog clause which has the same meaning is:

```
offspring( Y, X) :- parent( X, Y).
```

This clause can also be read as:

For all X and Y,
  if X is a parent of Y then
  Y is an offspring of X.

Prolog clauses such as:

```
offspring( Y, X) :- parent( X, Y).
```

are called *rules*. There is an important difference between facts and rules. A fact like:

```
parent( tom, liz).
```

is something that is always, unconditionally, true. On the other hand, rules specify things that are true if some condition is satisfied. Therefore we say that rules have:

- a condition part (the right-hand side of the rule) and
- a conclusion part (the left-hand side of the rule).

The conclusion part is also called the *head* of a clause and the condition part the *body* of a clause. For example:

$$\underbrace{\texttt{offspring(Y, X)}}_{\text{head}} \texttt{ :- } \underbrace{\texttt{parent(X, Y)}}_{\text{body}}.$$

If the condition parent(X, Y) is true then a logical consequence of this is offspring(Y, X).

How rules are actually used by Prolog is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom:

```
?- offspring( liz, tom).
```

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to any objects X and Y; therefore it can also be applied to such particular objects as liz and tom. To apply the rule to liz and tom, Y has to be substituted with liz, and X with tom. We say that the variables X and Y become instantiated to:

X = tom and Y = liz

After the instantiation we have obtained a special case of our general rule. The special case is:

```
offspring( liz, tom) :- parent( tom, liz).
```

The condition part has become:

```
parent( tom, liz)
```

Now Prolog tries to find out whether the condition part is true. So the initial goal:

```
offspring( liz, tom)
```

has been replaced with the subgoal:

```
parent( tom, liz)
```

This (new) goal happens to be trivial as it can be found as a fact in our program. This means that the conclusion part of the rule is also true, and Prolog will answer the question with yes.

Let us now add more family relations to our example program. The specification of the mother relation can be based on the following logical statement:

For all X and Y,
  X is the mother of Y if
  X is a parent of Y and
  X is a female.

This is translated into Prolog as the following rule:

```
mother( X, Y) :- parent( X, Y), female( X).
```

A comma between two conditions indicates the conjunction of the conditions, meaning that *both* conditions have to be true.

Relations such as parent, offspring and mother can be illustrated by diagrams such as those in Figure 1.3. These diagrams conform to the following conventions. Nodes in the graphs correspond to objects – that is, arguments of relations. Arcs between nodes correspond to binary (or two-place) relations. The arcs are oriented so as to point from the first argument of the relation to the second argument. Unary relations are indicated in the diagrams by simply marking the corresponding objects

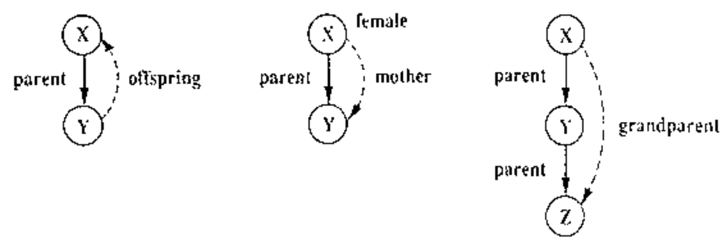


Figure 1.3 Definition graphs for the relations offspring, mother and grandparent in terms of other relations.

with the name of the relation. The relations that are being defined are represented by dashed arcs. So each diagram should be understood as follows: if the relations shown by solid arcs hold, then the relation shown by a dashed arc also holds. The grandparent relation can be, according to Figure 1.3, immediately written in Prolog as:

```
grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
```

At this point it will be useful to make a comment on the layout of our programs. Prolog gives us almost full freedom in choosing the layout of the program. So we can insert spaces and new lines as it best suits our taste. In general we want to make our programs look nice and tidy, and, above all, easy to read. To this end we will often choose to write the head of a clause and each goal of the body on a separate line. When doing this, we will indent goals in order to make the difference between the head and the goals more visible. For example, the grandparent rule would be, according to this convention, written as follows:

```
grandparent( X, Z) :-
  parent( X, Y),
  parent( Y, Z).
```

Figure 1.4 illustrates the sister relation:

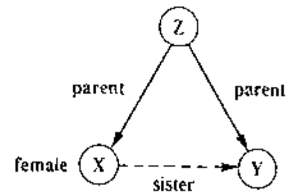


Figure 1.4 Defining the sister relation.

For any X and Y,
X is a sister of Y if
(1) both X and Y have the same parent, and
(2) X is a female.

The graph in Figure 1.4 can be translated into Prolog as:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X).
```

Notice the way in which the requirement 'both X and Y have the same parent' has been expressed. The following logical formulation was used: some Z must be a parent of X, and this *same* Z must be a parent of Y. An alternative, but less elegant way would be to say: Z1 is a parent of X, and Z2 is a parent of Y, and Z1 is equal to Z2.

We can now ask:

```
?- sister( ann, pat).
```

The answer will be 'yes', as expected (see Figure 1.1). Therefore we might conclude that the sister relation, as defined, works correctly. There is, however, a rather subtle flaw in our program, which is revealed if we ask the question Who is Pat's sister?:

```
?- sister( X, pat).
```

Prolog will find two answers, one of which may come as a surprise:

```
X = ann;
X = pat
```

So, Pat is a sister to herself?! This is probably not what we had in mind when defining the sister relation. However, according to our rule about sisters Prolog's answer is perfectly logical. Our rule about sisters does not mention that X and Y must not be the same if X is to be a sister of Y. As this is not required Prolog (rightfully) assumes that X and Y can be the same, and will as a consequence find that any female who has a parent is a sister of herself.

To correct our rule about sisters we have to add that X and Y must be different. We will see in later chapters how this can be done in several ways, but for the moment we will assume that a relation different is already known to Prolog, and that:

```
different( X, Y)
```

is satisfied if and only if X and Y are not equal. An improved rule for the sister relation can then be:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X),
  different( X, Y).
```


Some important points of this section are:

- Prolog programs can be extended by simply adding new clauses.
- Prolog clauses are of three types: *facts*, *rules* and *questions*.
- *Facts* declare things that are always, unconditionally true.
- *Rules* declare things that are true depending on a given condition.
- By means of *questions* the user can ask the program what things are true.
- Prolog clauses consist of the *head* and the *body*. The body is a list of *goals* separated by commas. Commas are understood as conjunctions.
- Facts are clauses that have a head and the empty body. Questions only have the body. Rules have the head and the (non-empty) body.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes *instantiated*.
- Variables are assumed to be universally quantified and are read as 'for all'. Alternative readings are, however, possible for variables that appear only in the body. For example:

  ```
  hasachild( X) :- parent( X, Y).
  ```

  can be read in two ways:

  (a) *For all* X and Y,
  if X is a parent of Y then
  X has a child.

  (b) *For all* X,
  X has a child if
  there is *some* Y such that X is a parent of Y.

Exercises

1.3 Translate the following statements into Prolog rules:

(a) Everybody who has a child is happy (introduce a one-argument relation happy).

(b) For all X, if X has a child who has a sister then X has two children (introduce new relation hastwochildren).

1.4 Define the relation grandchild using the parent relation. Hint: It will be similar to the grandparent relation (see Figure 1.3).

1.5 Define the relation aunt(X, Y) in terms of the relations parent and sister. As an aid you can first draw a diagram in the style of Figure 1.3 for the aunt relation.

1.3 Recursive rules

Let us add one more relation to our family program, the predecessor relation. This relation will be defined in terms of the parent relation. The whole definition can be expressed with two rules. The first rule will define the direct (immediate) predecessors and the second rule the indirect predecessors. We say that some X is an indirect predecessor of some Z if there is a parentship chain of people between X and Z, as illustrated in Figure 1.5. In our example of Figure 1.1, Tom is a direct predecessor of Liz and an indirect predecessor of Pat.

The first rule is simple and can be formulated as:

For all X and Z,
  X is a predecessor of Z if
  X is a parent of Z.

This is straightforwardly translated into Prolog as:

```
predecessor( X, Z) :-
  parent( X, Z).
```

The second rule, on the other hand, is more complicated because the chain of parents may present some problems. One attempt to define indirect predecessors could be as shown in Figure 1.6. According to this, the predecessor relation would be defined by a set of clauses as follows:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  parent( Y, Z).
```

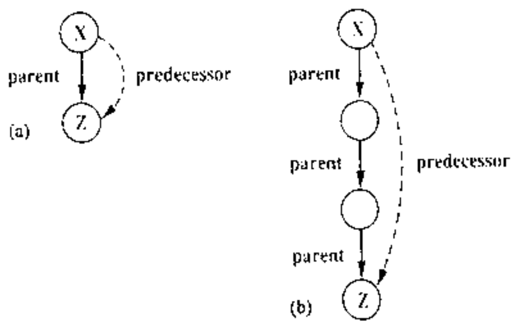


Figure 1.5 Examples of the predecessor relation: (a) X is a direct predecessor of Z; (b) X is an indirect predecessor of Z.

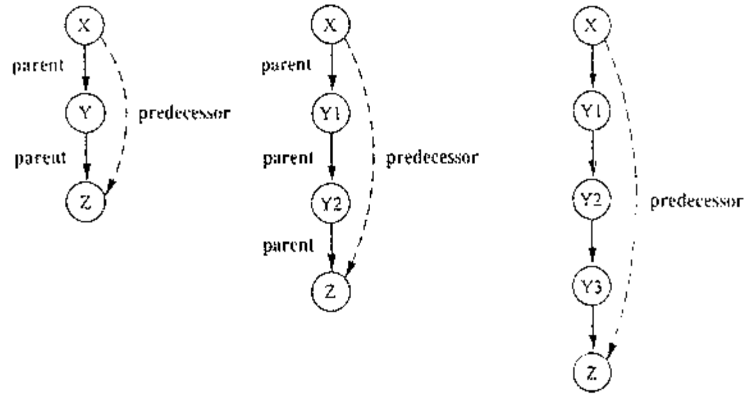


Figure 1.6 Predecessor-successor pairs at various distances.

```
predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Z).
predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Y3),
  parent( Y3, Z).
...
```

This program is lengthy and, more importantly, it only works to some extent. It would only discover predecessors to a certain depth in a family tree because the length of the chain of people between the predecessor and the successor would be limited according to the length of our predecessor clauses.

There is, however, an elegant and correct formulation of the predecessor relation: it will be correct in the sense that it will work for predecessors at any depth. The key idea is to define the predecessor relation in terms of itself. Figure 1.7 illustrates the idea:

For all X and Z,
  X is a predecessor of Z if
  there is a Y such that
  (1) X is a parent of Y and
  (2) Y is a predecessor of Z.

A Prolog clause with the above meaning is:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

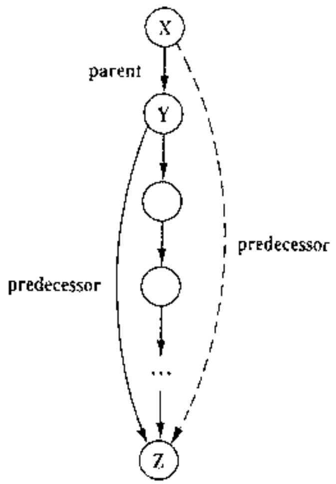


Figure 1.7 Recursive formulation of the predecessor relation.

We have thus constructed a complete program for the predecessor relation, which consists of two rules: one for direct predecessors and one for indirect predecessors. Both rules are rewritten together here:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

The key to this formulation was the use of predecessor itself in its definition. Such a definition may look surprising in view of the question: When defining something, can we use this same thing that has not yet been completely defined? Such definitions are, in general, called *recursive* definitions. Logically, they are perfectly correct and understandable, which is also intuitively obvious if we look at Figure 1.7. But will the Prolog system be able to use recursive rules? It turns out that Prolog can indeed very easily use recursive definitions. Recursive programming is, in fact, one of the fundamental principles of programming in Prolog. It is not possible to solve tasks of any significant complexity in Prolog without the use of recursion.

Going back to our program, we can ask Prolog: Who are Pam's successors? That is: Who is a person that has Pam as his or her predecessor?

```
?- predecessor( pam, X).

X = bob;

X = ann;

X = pat;

X = jim
```


Prolog's answers are, of course, correct and they logically follow from our definition of the predecessor and the parent relation. There is, however, a rather important question: *How* did Prolog actually use the program to find these answers?

An informal explanation of how Prolog does this is given in the next section. But first let us put together all the pieces of our family program, which was extended gradually by adding new facts and rules. The final form of the program is shown in Figure 1.8. Looking at Figure 1.8, two further points are in order here: the

```
parent( pam, bob).                 % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).                      % Pam is female
male( tom).                        % Tom is male
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X) :-                % Y is an offspring of X if
  parent( X, Y).                   % X is a parent of Y

mother( X, Y) :-                   % X is the mother of Y if
  parent( X, Y),                   % X is a parent of Y and
  female( X).                      % X is female

grandparent( X, Z) :-              % X is a grandparent of Z if
  parent( X, Y),                   % X is a parent of Y and
  parent( Y, Z).                   % Y is a parent of Z

sister( X, Y) :-                   % X is a sister of Y if
  parent( Z, X),
  parent( Z, Y),                   % X and Y have the same parent and
  female( X),                      % X is female and
  different( X, Y).                % X and Y are different

predecessor( X, Z) :-              % Rule pr1: X is a predecessor of Z
  parent( X, Z).

predecessor( X, Z) :-              % Rule pr2: X is a predecessor of Z
  parent( X, Y),
  predecessor( Y, Z).
```

Figure 1.8 The family program.

first will introduce the term 'procedure', the second will be about comments in programs.

The program in Figure 1.8 defines several relations – parent, male, female, predecessor, etc. The predecessor relation, for example, is defined by two clauses. We say that these two clauses are *about* the predecessor relation. Sometimes it is convenient to consider the whole set of clauses about the same relation. Such a set of clauses is called a *procedure*.

In Figure 1.8, the two rules about the predecessor relation have been distinguished by the names 'pr1' and 'pr2', added as *comments* to the program. These names will be used later as references to these rules. Comments are, in general, ignored by the Prolog system. They only serve as a further clarification to the person who reads the program. Comments are distinguished in Prolog from the rest of the program by being enclosed in special brackets '/*' and '*/'. Thus comments in Prolog look like this:

```
/* This is a comment */
```

Another method, more practical for short comments, uses the percent character '%'. Everything between '%' and the end of the line is interpreted as a comment:

```
% This is also a comment
```

Exercise

1.6 Consider the following alternative definition of the predecessor relation:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( Y, Z),
  predecessor( X, Y).
```

Does this also seem to be a correct definition of predecessors? Can you modify the diagram of Figure 1.7 so that it would correspond to this new definition?

1.4 How Prolog answers questions

This section gives an informal explanation of *how* Prolog answers questions. A question to Prolog is always a sequence of one or more goals. To answer a question, Prolog tries to satisfy all the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, Prolog also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If Prolog cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then Prolog's answer to the question will be 'no'.

An appropriate view of the interpretation of a Prolog program in mathematical terms is then as follows: Prolog accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem – that is, to demonstrate that it can be logically derived from the axioms.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.
Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man then X is fallible.

Accordingly, the example can be translated into Prolog as follows:

```
fallible( X) :- man( X).        % All men are fallible
man( socrates).                 % Socrates is a man
?- fallible( socrates).         % Socrates is fallible?
yes
```

A more complicated example from the family program of Figure 1.8 is:

```
?- predecessor( tom, pat).
```

We know that parent(bob, pat) is a fact. Using this fact and rule *pr1* we can conclude predecessor(bob, pat). This is a *derived* fact: it cannot be found explicitly in our program, but it can be derived from facts and rules in the program. An inference step, such as this, can be written in a more compact form as:

parent(bob, pat) ==> predecessor(bob, pat)

This can be read: from parent(bob, pat) it follows that predecessor(bob, pat), by rule *pr1*. Further, we know that parent(tom, bob) is a fact. Using this fact and the derived fact predecessor(bob, pat) we can conclude predecessor(tom, pat), by rule *pr2*. We have thus shown that our goal statement predecessor(tom, pat) is true. This whole inference process of two steps can be written as:

parent(bob, pat) ==> predecessor(bob, pat)

parent(tom, bob) *and* predecessor(bob, pat) ==> predecessor(tom, pat)

We have thus shown *what* can be a sequence of steps that satisfy a goal – that is, make it clear that the goal is true. Let us call this a proof sequence. We have not, however, shown *how* the Prolog system actually finds such a proof sequence.

Prolog finds the proof sequence in the inverse order to that which we have just used. Instead of starting with simple facts given in the program, Prolog starts with the goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts. Given the question:

```
?- predecessor( tom, pat).
```

Prolog will try to satisfy this goal. In order to do so it will try to find a clause in the program from which the above goal could immediately follow. Obviously, the only clauses relevant to this end are *pr1* and *pr2*. These are the rules about the predecessor relation. We say that the heads of these rules *match* the goal.

The two clauses, *pr1* and *pr2*, represent two alternative ways for Prolog to proceed. Prolog first tries that clause which appears first in the program:

```
predecessor( X, Z) :- parent( X, Z).
```

Since the goal is predecessor(tom, pat), the variables in the rule must be instantiated as follows:

```
X = tom, Z = pat
```

The original goal predecessor(tom, pat) is then replaced by a new goal:

```
parent( tom, pat)
```

This step of using a rule to transform a goal into another goal, as above, is graphically illustrated in Figure 1.9. There is no clause in the program whose head matches the goal parent(tom, pat), therefore this goal fails. Now Prolog *backtracks* to the original goal in order to try an alternative way to derive the top goal predecessor(tom, pat). The rule *pr2* is thus tried:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

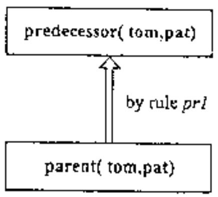


Figure 1.9 The first step of the execution. The top goal is true if the bottom goal is true.

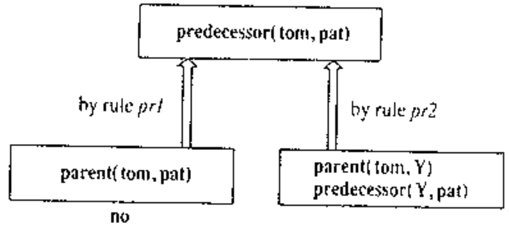


Figure 1.10 Execution trace continued from Figure 1.9.

As before, the variables X and Z become instantiated as:

```
X = tom, Z = pat
```

But Y is not instantiated yet. The top goal predecessor(tom, pat) is replaced by two goals:

```
parent( tom, Y),
predecessor( Y, pat)
```

This executional step is shown in Figure 1.10, which is an extension to the situation we had in Figure 1.9.

Being now faced with *two* goals, Prolog tries to satisfy them in the order in which they are written. The first one is easy as it matches one of the facts in the program. The matching forces Y to become instantiated to bob. Thus the first goal has been satisfied, and the remaining goal has become:

```
predecessor( bob, pat)
```

To satisfy this goal the rule *pr1* is used again. Note that this (second) application of the same rule has nothing to do with its previous application. Therefore, Prolog uses a new set of variables in the rule each time the rule is applied. To indicate this we shall rename the variables in rule *pr1* for this application as follows:

```
predecessor( X', Z') :-
  parent( X', Z').
```

The head has to match our current goal predecessor(bob, pat). Therefore:

```
X' = bob, Z' = pat
```

The current goal is replaced by:

```
parent( bob, pat)
```

This goal is immediately satisfied because it appears in the program as a fact. This completes the execution trace, which is graphically shown in Figure 1.11.

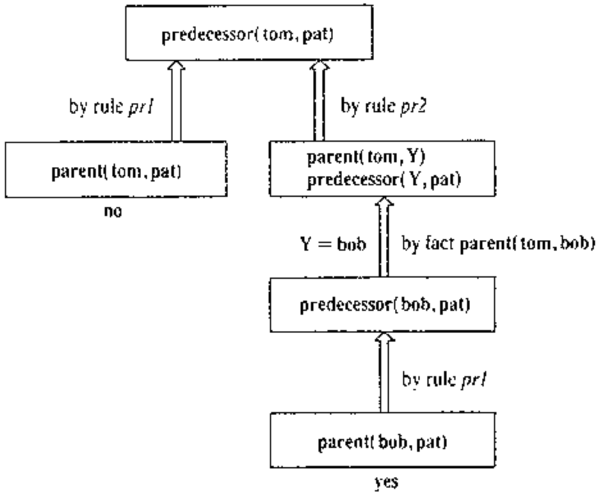


Figure 1.11 The complete execution trace to satisfy the goal predecessor(tom, pat). The right-hand branch proves the goal is satisfiable.

The graphical illustration of the execution trace in Figure 1.11 has the form of a tree. The nodes of the tree correspond to goals, or to lists of goals that are to be satisfied. The arcs between the nodes correspond to the application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled 'yes'. A leaf is labelled 'yes' if it is a simple fact. The execution of Prolog programs is the searching for such paths. During the search Prolog may enter an unsuccessful branch. When Prolog discovers that a branch fails it automatically *backtracks* to the previous node and tries to apply an alternative clause at that node.

Exercise

1.7 Try to understand how Prolog derives answers to the following questions, using the program of Figure 1.8. Try to draw the corresponding derivation diagrams in the style of Figures 1.9 to 1.11. Will any backtracking occur at particular questions?

(a) ?- parent(pam, bob).

(b) ?- mother(pam, bob).

(c) ?- grandparent(pam, ann).

(d) ?- grandparent(bob, jim).

1.5 Declarative and procedural meaning of programs

In our examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of Prolog programs; namely,

- the *declarative meaning* and
- the *procedural meaning*.

The declarative meaning is concerned only with the *relations* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how the relations are actually evaluated by the Prolog system.

The ability of Prolog to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the Prolog system itself.

This declarative approach indeed often makes programming in Prolog easier than in typical procedurally oriented programming languages such as C or Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about Prolog programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

Summary

- Prolog programming consists of defining relations and querying about relations.
- A program consists of *clauses*. These are of three types: *facts*, *rules* and *questions*.
- A relation can be specified by *facts*, simply stating the n-tuples of objects that satisfy the relation, or by stating *rules* about the relation.

- A *procedure* is a set of clauses about the same relation.
- Querying about relations, by means of *questions*, resembles querying a database. Prolog's answer to a question consists of a set of objects that satisfy the question.
- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the Prolog system and is, in principle, hidden from the user.
- Two types of meaning of Prolog programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.
- The following concepts have been introduced in this chapter:

  clause, fact, rule, question
  the head of a clause, the body of a clause
  recursive rule, recursive definition
  procedure
  atom, variable
  instantiation of a variable
  goal
  goal is satisfiable, goal succeeds
  goal is unsatisfiable, goal fails
  backtracking
  declarative meaning, procedural meaning

References

Various implementations of Prolog use different syntactic conventions. However, most of them follow the tradition of the so-called Edinburgh syntax (also called DEC-10 syntax, established by the historically influential implementation of Prolog for the DEC-10 computer; Pereira *et al.* 1978; Bowen 1981). The Edinburgh syntax also forms the basis of the ISO international standard for Prolog ISO/IEC 13211-1 (Deransart *et al.* 1996). Major Prolog implementations now largely comply with the standard. In this book we use a subset of the standard syntax, with some small and insignificant differences. In rare cases of such differences, there is a note to this effect at an appropriate place.

Bowen, D.L. (1981) *DECsystem-10 Prolog User's Manual*. University of Edinburgh: Department of Artificial Intelligence.
Deransart, P., Ed-Bdali, A. and Ceroni, L. (1996) *Prolog: The Standard*. Berlin: Springer-Verlag.
Pereira, L.M., Pereira, F. and Warren, D.H.D. (1978) *User's Guide to DECsystem-10 Prolog*. University of Edinburgh: Department of Artificial Intelligence.

chapter 2

Syntax and Meaning of Prolog Programs

This chapter gives a systematic treatment of the syntax and semantics of basic concepts of Prolog, and introduces structured data objects. The topics included are:

- simple data objects (atoms, numbers, variables)
- structured objects
- matching as the fundamental operation on objects
- declarative (or non-procedural) meaning of a program
- procedural meaning of a program
- relation between the declarative and procedural meanings of a program
- altering the procedural meaning by reordering clauses and goals.

Most of these topics have already been reviewed in Chapter 1. Here the treatment will become more formal and detailed.

2.1 Data objects

Figure 2.1 shows a classification of data objects in Prolog. The Prolog system recognizes the type of an object in the program by its syntactic form. This is possible because the syntax of Prolog specifies different forms for each type of data object. We have already seen a method for distinguishing between atoms and variables in Chapter 1: variables start with upper-case letters whereas atoms start with lower-case letters. No additional information (such as data-type declaration) has to be communicated to Prolog in order to recognize the type of an object.

2.1.1 Atoms and numbers

In Chapter 1 we have seen some simple examples of atoms and variables. In general, however, they can take more complicated forms – that is, strings of the following characters:

- upper-case letters A, B, ..., Z
- lower-case letters a, b, ..., z
- digits 0, 1, 2, ..., 9
- special characters such as + - * / < > = : . & _ ~

Atoms can be constructed in three ways:

(1) Strings of letters, digits and the underscore character, '_', starting with a lower-case letter:

```
anna
nil
x25
x_25
x_25AB
x_
x___y
alpha_beta_procedure
miss_Jones
sarah_jones
```

(2) Strings of special characters:

```
<--->
======>
...
.:.
::=
```

When using atoms of this form, some care is necessary because some strings of special characters already have a predefined meaning; an example is ':-'.

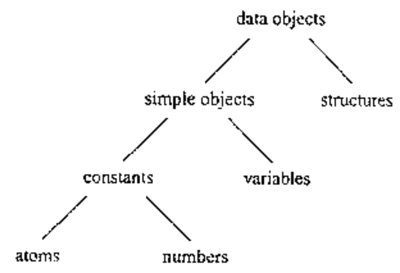


Figure 2.1 Data objects in Prolog.

(3) Strings of characters enclosed in single quotes. This is useful if we want, for example, to have an atom that starts with a capital letter. By enclosing it in quotes we make it distinguishable from variables:

```
'Tom'
'South_America'
'Sarah Jones'
```

Numbers used in Prolog include integer numbers and real numbers. The syntax of integers is simple, as illustrated by the following examples:

1   1313   0   -97

Not all integer numbers can be represented in a computer, therefore the range of integers is limited to an interval between some smallest and some largest number permitted by a particular Prolog implementation.

We will assume the simple syntax of real numbers, as shown by the following examples:

3.14   -0.0035   100.2

Real numbers are not very heavily used in typical Prolog programming. The reason for this is that Prolog is primarily a language for symbolic, non-numeric computation. In symbolic computation, integers are often used, for example, to count the number of items in a list; but there is typically less need for real numbers.

Apart from this lack of necessity to use real numbers in typical Prolog applications, there is another reason for avoiding real numbers. In general, we want to keep the meaning of programs as neat as possible. The introduction of real numbers somewhat impairs this neatness because of numerical errors that arise due to rounding when doing arithmetic. For example, the evaluation of the expression

10000 + 0.0001 - 10000

may result in 0 instead of the correct result 0.0001.

2.1.2 Variables

Variables are strings of letters, digits and underscore characters. They start with an upper-case letter or an underscore character:

```
X
Result
Object2
Participant_list
ShoppingList
_x23
_23
```

When a variable appears in a clause once only, we do not have to invent a name for it. We can use the so-called 'anonymous' variable, which is written as a single underscore character. For example, let us consider the following rule:

```
hasachild( X) :- parent( X, Y).
```

This rule says: for all X, X has a child if X is a parent of some Y. We are defining the property hasachild which, as it is meant here, does not depend on the name of the child. Thus, this is a proper place in which to use an anonymous variable. The clause above can thus be rewritten:

```
hasachild( X) :- parent( X, _).
```

Each time a single underscore character occurs in a clause it represents a new anonymous variable. For example, we can say that there is somebody who has a child if there are two objects such that one is a parent of the other:

```
somebody_has_child :- parent( _, _).
```

This is equivalent to:

```
somebody_has_child :- parent( X, Y).
```

But this is, of course, quite different from:

```
somebody_has_child :- parent( X, X).
```

If the anonymous variable appears in a question clause then its value is not output when Prolog answers the question. If we are interested in people who have children, but not in the names of the children, then we can simply ask:

```
?- parent( X, _).
```

The *lexical scope* of variable names is one clause. This means that, for example, if the name X15 occurs in two clauses, then it signifies two different variables. But each occurrence of X15 within the same clause means the same variable. The situation is different for constants: the same atom always means the same object in any clause – that is, throughout the whole program.

2.1.3 Structures

Structured objects (or simply *structures*) are objects that have several components. The components themselves can, in turn, be structures. For example, the date can be viewed as a structure with three components: day, month, year. Although composed of several components, structures are treated in the program as single objects. In order to combine the components into a single object we have to choose a *functor*. A suitable functor for our example is date. Then the date 1 May 2001 can be written as:

```
date( 1, may, 2001)
```

(see Figure 2.2).

All the components in this example are constants (two integers and one atom). Components can also be variables or other structures. Any day in May can be represented by the structure:

```
date( Day, may, 2001)
```

Note that Day is a variable and can be instantiated to any object at some later point in the execution.

This method for data structuring is simple and powerful. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation.

Syntactically, all data objects in Prolog are *terms*. For example,

```
may
```

and

```
date( 1, may, 2001)
```

are terms.

All structured objects can be pictured as trees (see Figure 2.2 for an example). The root of the tree is the functor, and the offsprings of the root are the components. If a component is also a structure then it is a subtree of the tree that corresponds to the whole structured object.

Our next example will show how structures can be used to represent some simple geometric objects (see Figure 2.3). A point in two-dimensional space is defined by its

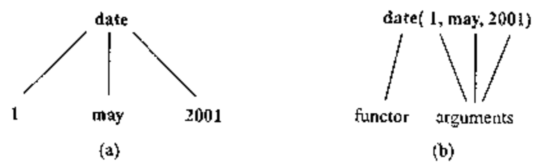


Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.

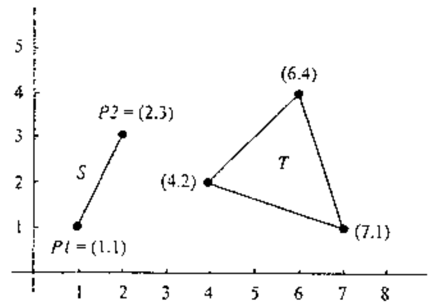


Figure 2.3 Some simple geometric objects.

two coordinates; a line segment is defined by two points; and a triangle can be defined by three points. Let us choose the following functors:

point	for points,
seg	for line segments, and
triangle	for triangles.

Then the objects in Figure 2.3 can be represented as follows:

```
P1 = point(1,1)
P2 = point(2,3)
S = seg( P1, P2) = seg( point(1,1), point(2,3) )
T = triangle( point(4,2), point(6,4), point(7,1) )
```

The corresponding tree representation of these objects is shown in Figure 2.4. In general, the functor at the root of the tree is called the *principal functor* of the term.

If in the same program we also had points in three-dimensional space then we could use another functor, point3, say, for their representation:

```
point3( X, Y, Z)
```

We can, however, use the same name, point, for points in both two and three dimensions, and write for example:

```
point( X1, Y1)   and   point( X, Y, Z)
```

If the same name appears in the program in two different roles, as is the case for point above, the Prolog system will recognize the difference by the number of arguments, and will interpret this name as two functors: one of them with two arguments and the other one with three arguments. This is so because each functor is defined by two things:

(1) the name, whose syntax is that of atoms;

(2) the *arity* – that is, the number of arguments.

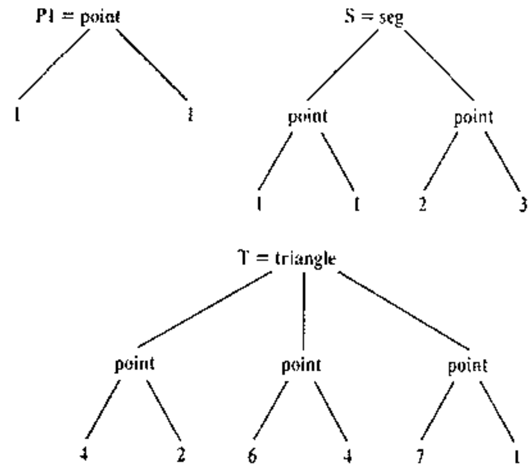


Figure 2.4 Tree representation of the objects in Figure 2.3.

As already explained, all structured objects in Prolog are trees, represented in the program by terms. We will study two more examples to illustrate how naturally complicated data objects can be represented by Prolog terms. Figure 2.5 shows the tree structure that corresponds to the arithmetic expression:

$(a + b) * (c - 5)$

According to the syntax of terms introduced so far this can be written, using the symbols '*', '+' and '–' as functors, as follows:

```
*( +( a, b), -( c, 5) )
```

This is, of course, a legal Prolog term; but this is not the form that we would normally like to have. We would normally prefer the usual, infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols '*', '+' and '–' are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

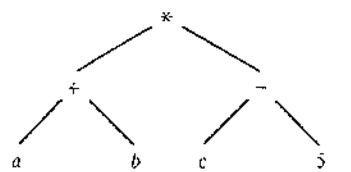


Figure 2.5 A tree structure that corresponds to the arithmetic expression $(a + b) * (c - 5)$.

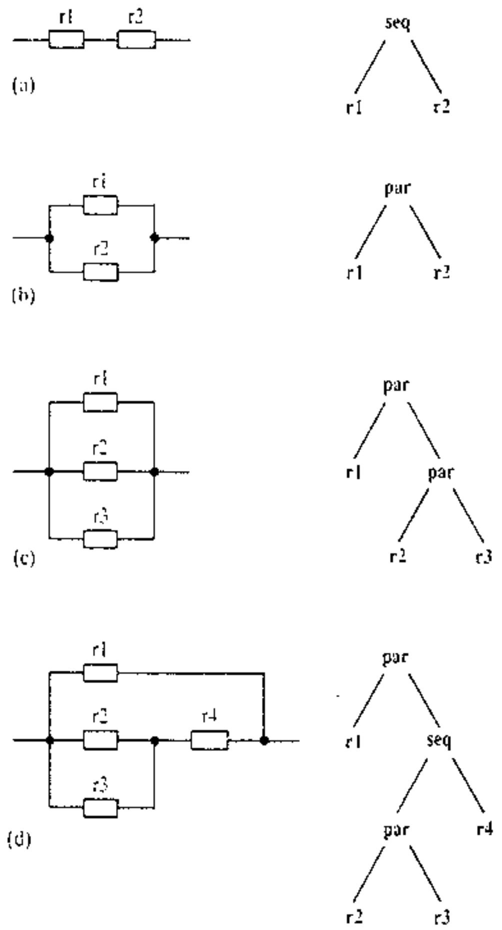


Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.

As the last example we consider some simple electric circuits shown in Figure 2.6. The right-hand side of the figure shows the tree representation of these circuits. The atoms r1, r2, r3 and r4 are the names of the resistors. The functors par and seq denote the parallel and the sequential compositions of resistors respectively. The corresponding Prolog terms are:


```
seq( r1, r2)
par( r1, r2)
par( r1, par( r2, r3) )
par( r1, seq( par( r2, r3), r4) )
```

Exercises

2.1 Which of the following are syntactically correct Prolog objects? What kinds of object are they (atom, number, variable, structure)?

(a) Diana
(b) diana
(c) 'Diana'
(d) _diana
(e) 'Diana goes south'
(f) goes(diana, south)
(g) 45
(h) 5(X, Y)
(i) +(north, west)
(j) three(Black(Cats))

2.2 Suggest a representation for rectangles, squares and circles as structured Prolog objects. Use an approach similar to that in Figure 2.4. For example, a rectangle can be represented by four points (or maybe three points only). Write some example terms that represent some concrete objects of these types using the suggested representation.

2.2 Matching

In the previous section we have seen how terms can be used to represent complex data objects. The most important operation on terms is *matching*. Matching alone can produce some interesting computation.

Given two terms, we say that they *match* if:

(1) they are identical, or
(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

For example, the terms date(D, M, 2001) and date(D1, may, Y1) match. One instantiation that makes both terms identical is:

- D is instantiated to D1
- M is instantiated to may
- Y1 is instantiated to 2001

This instantiation is more compactly written in the familiar form in which Prolog outputs results:

```
D = D1
M = may
Y1 = 2001
```

On the other hand, the terms date(D, M, 2001) and date(D1, M1, 1444) do not match, nor do the terms date(X, Y, Z) and point(X, Y, Z).

Matching is a process that takes as input two terms and checks whether they match. If the terms do not match we say that this process *fails*. If they do match then the process *succeeds* and it also instantiates the variables in both terms to such values that the terms become identical.

Let us consider again the matching of the two dates. The request for this operation can be communicated to the Prolog system by the following question, using the operator '=':

```
?- date( D, M, 2001) = date( D1, may, Y1).
```

We have already mentioned the instantiation D = D1, M = may, Y1 = 2001, which achieves the match. There are, however, other instantiations that also make both terms identical. Two of them are as follows:

```
D = 1
D1 = 1
M = may
Y1 = 2001

D = third
D1 = third
M = may
Y1 = 2001
```

These two instantiations are said to be *less general* than the first one because they constrain the values of the variables D and D1 more strongly than necessary. For making both terms in our example identical, it is only important that D and D1 have the same value, although this value can be anything. Matching in Prolog always results in the *most general* instantiation. This is the instantiation that commits the variables to the least possible extent, thus leaving the greatest possible freedom for further instantiations if further matching is required. As an example consider the following question:

```
?- date( D, M, 2001) = date( D1, may, Y1),
   date( D, M, 2001) = date( 15, M, Y).
```


To satisfy the first goal, Prolog instantiates the variables as follows:

```
D = D1
M = may
Y1 = 2001
```

After having satisfied the second goal, the instantiation becomes more specific as follows:

```
D = 15
D1 = 15
M = may
Y1 = 2001
Y = 2001
```

This example also shows that variables, during the execution of consecutive goals, typically become instantiated to increasingly more specific values.

The general rules to decide whether two terms, S and T, match are as follows:

(1) If S and T are constants then S and T match only if they are the same object.

(2) If S is a variable and T is anything, then they match, and S is instantiated to T. Conversely, if T is a variable then T is instantiated to S.

(3) If S and T are structures then they match only if
   (a) S and T have the same principal functor, and
   (b) all their corresponding components match.

   The resulting instantiation is determined by the matching of the components.

The last of these rules can be visualized by considering the tree representation of terms, as in the example of Figure 2.7. The matching process starts at the root (the principal functors). As both functors match, the process proceeds to the arguments where matching of the pairs of corresponding arguments occurs. So the whole matching process can be thought of as consisting of the following sequence of (simpler) matching operations:

```
triangle = triangle,
point(1,1) = X,
A = point(4,Y),
point(2,3) = point(2,Z).
```

The whole matching process succeeds because all the matchings in the sequence succeed. The resulting instantiation is:

```
X = point(1,1)
A = point(4,Y)
Z = 3
```

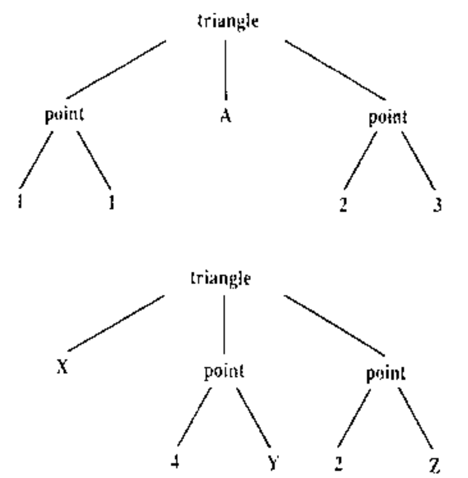


Figure 2.7 Matching triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z)).

The following example will illustrate how matching alone can be used for interesting computation. Let us return to the simple geometric objects of Figure 2.4, and define a piece of program for recognizing horizontal and vertical line segments. 'Vertical' is a property of segments, so it can be formalized in Prolog as a unary relation. Figure 2.8 helps to formulate this relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'horizontal' is similarly formulated, with only x and y interchanged. The following program, consisting of two facts, does the job:

```
vertical( seg( point(X,Y), point(X,Y1) ) ).
horizontal( seg( point(X,Y), point(X1,Y) ) ).
```

The following conversation is possible with this program:

```
?- vertical( seg( point(1,1), point(1,2) ) ).
yes
?- vertical( seg( point(1,1), point(2,Y) ) ).
no
?- horizontal( seg( point(1,1), point(2,Y) ) ).
Y = 1
```

The first question was answered 'yes' because the goal in the question matched one of the facts in the program. For the second question no match was possible. In the third question, Y was forced to become 1 by matching the fact about horizontal segments.

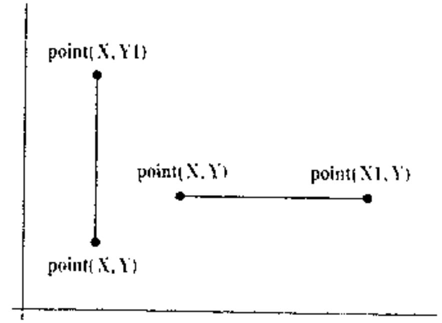


Figure 2.8 Illustration of vertical and horizontal line segments.

A more general question to the program is: Are there any vertical segments that start at the point (2,3)?

```
?- vertical( seg( point(2,3), P) ).
P = point(2,Y)
```

This answer means: Yes, any segment that ends at any point (2,Y), which means anywhere on the vertical line $x = 2$. It should be noted that Prolog's actual answer would probably not look as neat as above, but (depending on the Prolog implementation used) something like this:

```
P = point(2,_136)
```

This is, however, only a cosmetic difference. Here _136 is a variable that has not been instantiated. _136 is a legal variable name that the system has constructed during the execution. The system has to generate new names in order to rename the user's variables in the program. This is necessary for two reasons: first, because the same name in different clauses signifies different variables, and second, in successive applications of the same clause, its 'copy' with a new set of variables is used each time.

Another interesting question to our program is: Is there a segment that is both vertical and horizontal?

```
?- vertical( S), horizontal( S).
S = seg( point(X,Y), point(X,Y) )
```

This answer by Prolog says: Yes, any segment that is degenerated to a point has the property of being vertical and horizontal at the same time. The answer was, again, derived simply by matching. As before, some internally generated names may appear in the answer, instead of the variable names X and Y.

Will the following matching operations succeed or fail? If they succeed, what are the resulting instantiations of variables?

(a) point(A, B) = point(1, 2)

(b) point(A, B) = point(X, Y, Z)

(c) plus(2, 2) = 4

(d) +(2, D) = +(E, 2)

(e) triangle(point(−1,0), P2, P3) = triangle(P1, point(1,0), point(0,Y))

The resulting instantiation defines a family of triangles. How would you describe this family?

2.4 Using the representation for line segments as described in this section, write a term that represents any vertical line segment at $x = 5$.

2.5 Assume that a rectangle is represented by the term rectangle(P1, P2, P3, P4) where the P's are the vertices of the rectangle positively ordered. Define the relation:

```
regular( R)
```

which is true if R is a rectangle whose sides are vertical and horizontal.

2.3 Declarative meaning of Prolog programs

We have already seen in Chapter 1 that Prolog programs can be understood in two ways: declaratively and procedurally. In this and the next section we will consider a more formal definition of the declarative and procedural meanings of programs in basic Prolog. But first let us look at the difference between these two meanings again.

Consider a clause:

```
P :- Q, R.
```

where P, Q and R have the syntax of terms. Some alternative declarative readings of this clause are:

P is true if Q and R are true.
From Q and R follows P.

Two alternative procedural readings of this clause are:

To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.
To satisfy P, *first* satisfy Q and *then* R.

Thus the difference between the declarative readings and the procedural ones is that the latter do not only define the logical relations between the head of the clause and the goals in the body, but also the *order* in which the goals are processed.

Let us now formalize the declarative meaning.

The declarative meaning of programs determines whether a given goal is true, and if so, for what values of variables it is true. To precisely define the declarative meaning we need to introduce the concept of *instance* of a clause. An instance of a clause C is the clause C with each of its variables substituted by some term. A *variant* of a clause C is such an instance of the clause C where each variable is substituted by another variable. For example, consider the clause:

```
hasachild( X) :- parent( X, Y).
```

Two variants of this clause are:

```
hasachild( A) :- parent( A, B).
hasachild( X1) :- parent( X1, X2).
```

Instances of this clause are:

```
hasachild( peter) :- parent( peter, Z).
hasachild( barry) :- parent( barry, small(caroline) ).
```

Given a program and a goal G, the declarative meaning says:

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if:

(1) there is a clause C in the program such that
(2) there is a clause instance I of C such that
  (a) the head of I is identical to G, and
  (b) all the goals in the body of I are true.

This definition extends to Prolog questions as follows. In general, a question to the Prolog system is a *list* of goals separated by commas. A list of goals is true if *all* the goals in the list are true for the *same* instantiation of variables. The values of the variables result from the most general instantiation.

A comma between goals thus denotes the *conjunction* of goals: they *all* have to be true. But Prolog also accepts the *disjunction* of goals: *any one* of the goals in a disjunction has to be true. Disjunction is indicated by a semicolon. For example,

```
P :- Q; R.
```

is read: P is true if Q is true *or* R is true. The meaning of this clause is thus the same as the meaning of the following two clauses together:

```
P :- Q.
P :- R.
```

The comma binds stronger than the semicolon. So the clause:

```
P :- Q, R; S, T, U.
```

is understood as:

```
P :- ( Q, R); ( S, T, U).
```

and means the same as the clauses:

```
P :- Q, R.
P :- S, T, U.
```

Exercises

2.6 Consider the following program:

```
f( 1, one).

f( s(1), two).

f( s(s(1)), three).

f( s(s(s(X))), N) :-
  f( X, N).
```

How will Prolog answer the following questions? Whenever several answers are possible, give at least two.

(a) ?- f(s(1), A).

(b) ?- f(s(s(1)), two).

(c) ?- f(s(s(s(s(s(s(1)))))), C).

(d) ?- f(D, three).

2.7 The following program says that two people are relatives if

(a) one is a predecessor of the other, or

(b) they have a common predecessor, or

(c) they have a common successor:

```
relatives( X, Y) :-
  predecessor( X, Y).

relatives( X, Y) :-
  predecessor( Y, X).

relatives( X, Y) :-                 % X and Y have a common predecessor
  predecessor( Z, X),
  predecessor( Z, Y).

relatives( X, Y) :-                 % X and Y have a common successor
  predecessor( X, Z),
  predecessor( Y, Z).
```

Can you shorten this program by using the semicolon notation?

2.8 Rewrite the following program without using the semicolon notation.

```
translate( Number, Word) :-
  Number = 1, Word = one;
  Number = 2, Word = two;
  Number = 3, Word = three.
```

2.4 Procedural meaning

The procedural meaning specifies *how* Prolog answers questions. To answer a question means to try to satisfy a list of goals. They can be satisfied if the variables that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To 'execute goals' means: try to satisfy them.

Let us call this procedure execute. As shown in Figure 2.9, the inputs to and the outputs from this procedure are:

input: a program and a goal list
output: a success/failure indicator and an instantiation of variables

The meaning of the two output results is as follows:

(1) The success/failure indicator is 'yes' if the goals are satisfiable and 'no' otherwise. We say that 'yes' signals a *successful* termination and 'no' a *failure*.

(2) An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation.

In Chapter 1, we have in effect already discussed informally what procedure execute does, under the heading 'How Prolog answers questions'. What follows in the rest of this section is just a more formal and systematic description of this

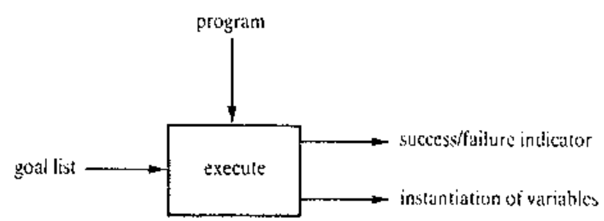


Figure 2.9 Input/output view of the procedure that executes a list of goals.

process, and can be skipped without seriously affecting the understanding of the rest of the book.

Particular operations in the goal execution process are illustrated by the example in Figure 2.10. It may be helpful to study Figure 2.10 before reading the following general description.

To execute a list of goals:

G1, G2, ..., Gm

the procedure execute does the following:

- If the goal list is empty then terminate with *success*.
- If the goal list is not empty then continue with (the following) operation called 'SCANNING'.
- *SCANNING*: Scan through the clauses in the program from top to bottom until the first clause, C, is found such that the head of C matches the first goal G1. If there is no such clause then terminate with *failure*.

  If there is such a clause C of the form

  H :- B1, ..., Bn.

  then rename the variables in C to obtain a variant C′ of C, such that C′ and the list G1, ..., Gm have no common variables. Let C′ be

  H′ :- B1′, ..., Bn′.

  Match G1 and H′; let the resulting instantiation of variables be S.

  In the goal list G1, G2, ..., Gm, replace G1 with the list B1′, ..., Bn′, obtaining a new goal list

  B1′, ..., Bn′, G2, ..., Gm

  (Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

  Substitute the variables in this new goal list with new values as specified in the instantiation S, obtaining another goal list

  B1″, ..., Bn″, G2′, ..., Gm′

- Execute (recursively with this same procedure) this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with the clause that immediately follows the clause C (C is the clause that was last used) and try to find a successful termination using some other clause.

PROGRAM

```
big( bear).                % Clause 1
big( elephant).            % Clause 2
small( cat).               % Clause 3

brown( bear).              % Clause 4
black( cat).               % Clause 5
gray( elephant).           % Clause 6

dark( Z) :-                % Clause 7: Anything black is dark
  black( Z).

dark( Z) :-                % Clause 8: Anything brown is dark
  brown( Z).
```

QUESTION

```
?- dark( X), big( X).      % Who is dark and big?
```

EXECUTION TRACE

(1) Initial goal list: dark(X), big(X).

(2) Scan the program from top to bottom looking for a clause whose head matches the first goal dark(X). Clause 7 found:

   dark(Z) :- black(Z).

   Replace the first goal by the instantiated body of clause 7, giving a new goal list:

   black(X), big(X)

(3) Scan the program to find a match with black(X). Clause 5 found: black(cat). This clause has no body, so the goal list, properly instantiated, shrinks to:

   big(cat)

(4) Scan the program for the goal big(cat). No clause found. Therefore backtrack to step (3) and undo the instantiation X = cat. Now the goal list is again:

   black(X), big(X)

   Continue scanning the program below clause 5. No clause found. Therefore backtrack to step (2) and continue scanning below clause 7. Clause 8 is found:

   dark(Z) :- brown(Z).

   Replace the first goal in the goal list by brown(X), giving:

   brown(X), big(X)

(5) Scan the program to match brown(X), finding brown(bear). This clause has no body, so the goal list shrinks to:

   big(bear)

(6) Scan the program and find clause big(bear). It has no body so the goal list shrinks to empty. This indicates successful termination, and the corresponding variable instantiation is:

   X = bear

Figure 2.10 An example to illustrate the procedural meaning of Prolog: a sample trace of the procedure execute.

This procedure is more compactly written in a Pascal-like notation in Figure 2.11.

Several additional remarks are in order here regarding the procedure execute as presented. First, it was not explicitly described how the final resulting instantiation of variables is produced. It is the instantiation S which led to a successful termination, and was possibly further refined by additional instantiations that were done in the nested recursive calls to execute.

Whenever a recursive call to execute fails, the execution returns to SCANNING, continuing at the program clause C that had been last used before. As the application of the clause C did not lead to a successful termination Prolog has to try an alternative clause to proceed. What effectively happens is that Prolog abandons this whole part of the unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all the variable instantiations that were done after that point are undone. This ensures that Prolog systematically examines all the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

We have already seen that even after a successful termination the user can force the system to backtrack to search for more solutions. In our description of execute this detail was left out.

Of course, in actual implementations of Prolog, several other refinements have to be added to execute. One of them is to reduce the amount of scanning through the program clauses to improve efficiency. So a practical Prolog implementation will not scan through all the clauses of the program, but will only consider the clauses about the relation in the current goal.

---

onsider the program in Figure 2.10 and simulate, in the style of Figure 2.10, olog's execution of the question:

```
?- big( X), dark( X).
```

ıpare your execution trace with that of Figure 2.10 when the question was ıtially the same, but with the goals in the order:

```
dark( X), big( X).
```

ch of the two cases does Prolog have to do more work before the answer is


```
procedure execute (Program, GoalList, Success);
Input arguments:
  Program: list of clauses
  GoalList: list of goals
Output argument:
  Success: truth value; Success will become true if GoalList is true with respect to Program
Local variables:
  Goal: goal
  OtherGoals: list of goals
  Satisfied: truth value
  MatchOK: truth value
  Instant: instantiation of variables
  H, H', B1, B1', ..., Bn, Bn': goals
Auxiliary functions:
  empty(L): returns true if L is the empty list
  head(L): returns the first element of list L
  tail(L): returns the rest of L
  append(L1,L2): appends list L2 at the end of list L1
  match(T1,T2,MatchOK,Instant): tries to match terms T1 and T2; if
    succeeds then MatchOK is true and Instant is the corresponding instantiation of variables
  substitute(Instant,Goals): substitutes variables in Goals according to instantiation Instant

begin
  if empty(GoalList) then Success := true
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      Satisfied := false;
      while not Satisfied and "more clauses in program" do
        begin
          Let next clause in Program be
            H :- B1, ..., Bn.
          Construct a variant of this clause
            H' :- B1', ..., Bn'.
          match(Goal,H',MatchOK,Instant);
          if MatchOK then
            begin
              NewGoals := append([B1',...,Bn'], OtherGoals);
              NewGoals := substitute(Instant,NewGoals);
              execute(Program,NewGoals,Satisfied)
            end
        end;
      Success := Satisfied
    end
end;
```

Figure 2.11 Executing Prolog goals.

2.5 Example: monkey and banana

The monkey and banana problem is used as a simple example of problem solving. Our Prolog program for this problem will show how the mechanisms of matching and backtracking can be used in such exercises. We will develop the program in the non-procedural way, and then study its procedural behaviour in detail. The program will be compact and illustrative.

We will use the following variation of the problem. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around (if it is already at the box) and grasp the banana if standing on the box directly under the banana. Can the monkey get the banana?

One important task in programming is that of finding a representation of the problem in terms of the programming language used. In our case we can think of the 'monkey world' as always being in some *state* that can change in time. The current state is determined by the positions of the objects. For example, the initial state of the world is determined by:

(1) Monkey is at door.

(2) Monkey is on floor.

(3) Box is at window.

(4) Monkey does not have banana.

It is convenient to combine all of these four pieces of information into one structured object. Let us choose the word 'state' as the functor to hold the four components together. Figure 2.12 shows the initial state represented as a structured object.

Our problem can be viewed as a one-person game. Let us now formalize the rules of the game. First, the goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is 'has':

```
state( _, _, _, has)
```

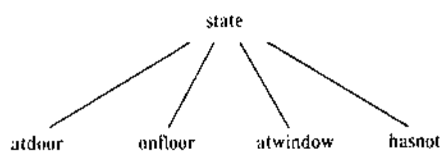


Figure 2.12 The initial state of the monkey world represented as a structured object. The four components are: horizontal position of monkey, vertical position of monkey, position of box, monkey has or has not banana.

Second, what are the allowed moves that change the world from one state to another? There are four types of moves:

(1) grasp banana,

(2) climb box,

(3) push box,

(4) walk around.

Not all moves are possible in every possible state of the world. For example, the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet. Such rules can be formalized in Prolog as a three-place relation named move:

```
move( State1, Move, State2)
```

The three arguments of the relation specify a move thus:

$$\text{State1} \xrightarrow[\text{Move}]{} \text{State2}$$

State1 is the state before the move, Move is the move executed and State2 is the state after the move.

The move 'grasp', with its necessary precondition on the state before the move, can be defined by the clause:

```
move( state( middle, onbox, middle, hasnot),     % Before move
      grasp,                                     % Move
      state( middle, onbox, middle, has) ).      % After move
```

This fact says that after the move the monkey has the banana, and he has remained on the box in the middle of the room.

In a similar way we can express the fact that the monkey on the floor can walk from any horizontal position Pos1 to any position Pos2. The monkey can do this regardless of the position of the box and whether it has the banana or not. All this can be defined by the following Prolog fact:

```
move( state( Pos1, onfloor, Box, Has),
      walk( Pos1, Pos2),                         % Walk from Pos1 to Pos2
      state( Pos2, onfloor, Box, Has) ).
```

Note that this clause says many things, including, for example:

- the move executed was 'walk from some position Pos1 to some position Pos2';
- the monkey is on the floor before and after the move;

- the box is at some point Box which remained the same after the move;
- the 'has banana' status Has remains the same after the move.

The clause actually specifies a whole set of possible moves because it is applicable to any situation that matches the specified state before the move. Such a specification is therefore sometimes also called a move *schema*. Using Prolog variables, such schemas can be easily programmed in Prolog.

The other two types of moves, 'push' and 'climb', can be similarly specified.

The main kind of question that our program will have to answer is: Can the monkey in some initial state State get the banana? This can be formulated as a predicate

```
canget( State)
```

where the argument State is a state of the monkey world. The program for canget can be based on two observations:

(1) For any state in which the monkey already has the banana, the predicate canget must certainly be true; no move is needed in this case. This corresponds to the Prolog fact:

```
canget( state( _, _, _, has) ).
```

(2) In other cases one or more moves are necessary. The monkey can get the banana in any state State1 if there is some move Move from State1 to some state State2, such that the monkey can then get the banana in state State2 (in zero or more moves). This principle is illustrated in Figure 2.13. A Prolog clause that corresponds to this rule is:

```
canget( State1) :-
  move( State1, Move, State2),
  canget( State2).
```

This completes our program, which is shown in Figure 2.14.

The formulation of canget is recursive and is similar to that of the predecessor relation of Chapter 1 (compare Figures 2.13 and 1.7). This principle is used in Prolog again and again.

We have developed our monkey and banana program in the non-procedural way. Let us now study its *procedural* behaviour by considering the following question

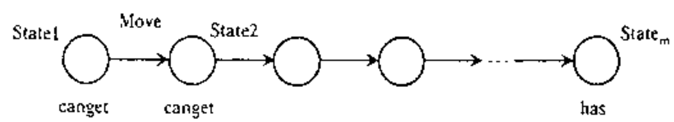


Figure 2.13 Recursive formulation of canget.


```
% move( State1, Move, State2): making Move in State1 results in State2;
%   a state is represented by a term:
%   state( MonkeyHorizontal, MonkeyVertical, BoxPosition, HasBanana)

move( state( middle, onbox, middle, hasnot),    % Before move
      grasp,                                    % Grasp banana
      state( middle, onbox, middle, has) ).     % After move

move( state( P, onfloor, P, H),
      climb,                                    % Climb box
      state( P, onbox, P, H) ).

move( state( P1, onfloor, P1, H),
      push( P1, P2),                            % Push box from P1 to P2
      state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
      walk( P1, P2),                            % Walk from P1 to P2
      state( P2, onfloor, B, H) ).

% canget( State): monkey can get banana in State

canget( state( _, _, _, has) ).                 % can 1: Monkey already has it

canget( State1) :-                              % can 2: Do some work to get it
      move( State1, Move, State2),              % Do something
      canget( State2).                          % Get it now
```

Figure 2.14 A program for the monkey and banana problem.

to the program:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

Prolog's answer is 'yes'. The process carried out by Prolog to reach this answer proceeds, according to the procedural semantics of Prolog, through a sequence of goal lists. It involves some search for the right moves among the possible alternative moves. At some point this search will take a wrong move leading to a dead branch. At this stage, backtracking will help it to recover. Figure 2.15 illustrates this search process.

To answer the question Prolog had to backtrack once only. A right sequence of moves was found almost straight away. The reason for this efficiency of the program was the order in which the clauses about the move relation occurred in the program. The order in our case (luckily) turned out to be quite suitable. However, less lucky orderings are possible. According to the rules of the game, the monkey could just as easily try to walk here or there without ever touching the box, or aimlessly push the box around. A more thorough investigation will reveal, as shown in the following section, that the ordering of clauses is, in the case of our program, in fact critical.

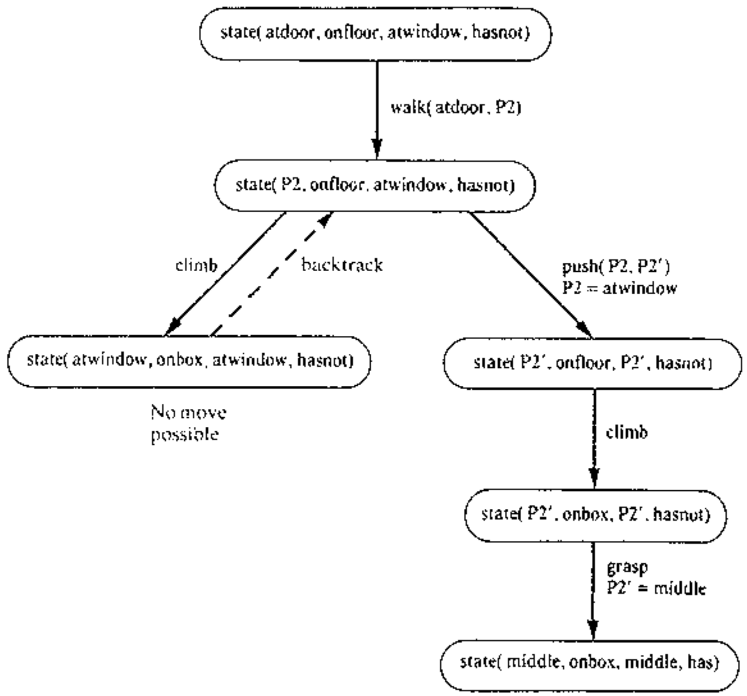


Figure 2.15 The monkey's search for the banana. The search starts at the top node and proceeds downwards, as indicated. Alternative moves are tried in the left-to-right order. Backtracking occurred once only.

2.6 Order of clauses and goals

2.6.1 Danger of indefinite looping

Consider the following clause:

```
p :- p.
```

This says that 'p is true if p is true'. This is declaratively perfectly correct, but procedurally is quite useless. In fact, such a clause can cause problems to Prolog. Consider the question:

```
?- p.
```


Using the clause above, the goal p is replaced by the same goal p; this will be in turn replaced by p, etc. In such a case Prolog will enter an infinite loop not noticing that no progress is being made.

This example is a simple way of getting Prolog to loop indefinitely. However, similar looping could have occurred in some of our previous example programs if we changed the order of clauses, or the order of goals in the clauses. It will be instructive to consider some examples.

In the monkey and banana program, the clauses about the move relation were ordered thus: grasp, climb, push, walk (perhaps 'unclimb' should be added for completeness). These clauses say that grasping is possible, climbing is possible, etc. According to the procedural semantics of Prolog, the order of clauses indicates that the monkey prefers grasping to climbing, climbing to pushing, etc. This order of preferences in fact helps the monkey to solve the problem. But what could happen if the order was different? Let us assume that the 'walk' clause appears first. The execution of our original goal of the previous section

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

would this time produce the following trace. The first four goal lists (with variables appropriately renamed) are the same as before:

(1) canget(state(atdoor, onfloor, atwindow, hasnot))

The second clause of canget ('can2') is applied, producing:

(2) move(state(atdoor, onfloor, atwindow, hasnot), M', S2'),
canget(S2')

By the move walk(atdoor, P2') we get:

(3) canget(state(P2', onfloor, atwindow, hasnot))

Using the clause 'can2' again the goal list becomes:

(4) move(state(P2', onfloor, atwindow, hasnot), M'', S2''),
canget(S2'')

Now the difference occurs. The first clause whose head matches the first goal above is now 'walk' (and not 'climb' as before). The instantiation is

S2'' = state(P2'', onfloor, atwindow, hasnot)

Therefore the goal list becomes:

(5) canget(state(P2'', onfloor, atwindow, hasnot))

Applying the clause 'can2' we obtain:

(6) move(state(P2'', onfloor, atwindow, hasnot), M''', S2'''),
canget(S2''')

Again, 'walk' is now tried first, producing:

```
(7)   canget( state( P2''', onfloor, atwindow, hasnot) )
```

Let us now compare the goals (3), (5) and (7). They are the same apart from one variable: this variable is, in turn, P', P'' and P'''. As we know, the success of a goal does not depend on particular names of variables in the goal. This means that from goal list (3) the execution trace shows no progress. We can see, in fact, that the same two clauses, 'can2' and 'walk', are used repetitively. The monkey walks around without ever trying to use the box. As there is no progress made this will (theoretically) go on for ever: Prolog will not realize that there is no point in continuing along this line.

This example shows Prolog trying to solve a problem in such a way that a solution is never reached, although a solution exists. Such situations are not unusual in Prolog programming. Infinite loops are, also, not unusual in other programming languages. What *is* unusual in comparison with other languages is that a Prolog program may be declaratively correct, but at the same time be procedurally incorrect in that it is not able to produce an answer to a question. In such cases Prolog may not be able to satisfy a goal because it tries to reach an answer by choosing a wrong path.

A natural question to ask at this point is: Can we not make some more substantial change to our program so as to drastically prevent any danger of looping? Or shall we always have to rely just on a suitable ordering of clauses and goals? As it turns out programs, especially large ones, would be too fragile if they just had to rely on some suitable ordering. There are several other methods that preclude infinite loops, and these are much more general and robust than the ordering method itself. These techniques will be used regularly later in the book, especially in those chapters that deal with path finding, problem solving and search.

2.6.2 Program variations through reordering of clauses and goals

Already in the example programs of Chapter 1 there was a latent danger of producing a cycling behaviour. Our program to specify the predecessor relation in Chapter 1 was:

```
predecessor( Parent, Child) :-
  parent( Parent, Child).

predecessor( Predecessor, Successor) :-
  parent( Predecessor, Child),
  predecessor( Child, Successor).
```

Let us analyze some variations of this program. All the variations will clearly have the same declarative meaning, but not the same procedural meaning. According to the declarative semantics of Prolog we can, without affecting the declarative meaning, change:

(1) the order of clauses in the program, and

(2) the order of goals in the bodies of clauses.

The predecessor procedure consists of two clauses, and one of them has two goals in the body. There are, therefore, four variations of this program, all with the same declarative meaning. The four variations are obtained by:

(1) swapping both clauses, and

(2) swapping the goals for each order of clauses.

The corresponding four procedures, called pred1, pred2, pred3 and pred4, are shown in Figure 2.16.

```
% Four versions of the predecessor program

% The original version

pred1( X, Z) :-
  parent( X, Z).

pred1( X, Z) :-
  parent( X, Y),
  pred1( Y, Z).

% Variation a: swap clauses of the original version

pred2( X, Z) :-
  parent( X, Y),
  pred2( Y, Z).

pred2( X, Z) :-
  parent( X, Z).

% Variation b: swap goals in second clause of the original version

pred3( X, Z) :-
  parent( X, Z).

pred3( X, Z) :-
  pred3( X, Y),
  parent( Y, Z).

% Variation c: swap goals and clauses of the original version

pred4( X, Z) :-
  pred4( X, Y),
  parent( Y, Z).

pred4( X, Z) :-
  parent( X, Z).
```

Figure 2.16 Four versions of the predecessor program.

There are important differences in the behaviour of these four declaratively equivalent procedures. To demonstrate these, consider the parent relation as shown in Figure 1.1 of Chapter 1. Now, what happens if we ask whether Tom is a predecessor of Pat using the four variations of the predecessor relation:

```
?- pred1( tom, pat).
yes
?- pred2( tom, pat).
yes
?- pred3( tom, pat).
yes
?- pred4( tom, pat).
```

In the last case Prolog cannot find the answer. This is manifested on the terminal by a Prolog message such as 'More core needed' or 'Stack overflow'.

Figure 1.11 in Chapter 1 showed the trace of pred1 (in Chapter 1 called predecessor) produced for the above question. Figure 2.17 shows the corresponding traces for pred2, pred3 and pred4. Figure 2.17(c) clearly shows that pred4 is hopeless, and Figure 2.17(a) indicates that pred2 is rather inefficient compared to pred1: pred2 does much more searching and backtracking in the family tree.

This comparison should remind us of a general practical heuristic in problem solving: it is usually best to try the simplest idea first. In our case, all the versions of the predecessor relation are based on two ideas:

(1) the simpler idea is to check whether the two arguments of the predecessor relation satisfy the parent relation;

(2) the more complicated idea is to find somebody 'between' both people (somebody who is related to them by the parent and predecessor relations).

Of the four variations of the predecessor relation, pred1 does simplest things first. On the contrary, pred4 always tries complicated things first. pred2 and pred3 are in between the two extremes. Even without a detailed study of the execution traces, pred1 should be preferred merely on the grounds of the rule 'try simple things first'. This rule will be in general a useful guide in programming.

Our four variations of the predecessor procedure can be further compared by considering the question: What types of questions can particular variations answer, and what types can they not answer? It turns out that pred1 and pred2 are both able to reach an answer for any type of question about predecessors; pred4 can never reach an answer; and pred3 sometimes can and sometimes cannot. One example in which pred3 fails is:

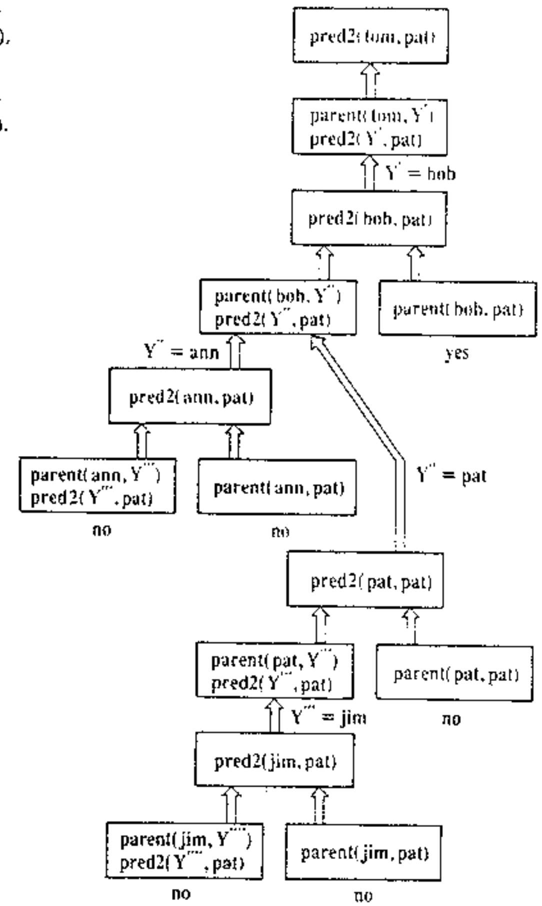
```
?- pred3( liz, jim).
```


This question again brings the system into an infinite sequence of recursive calls. Thus pred3 also cannot be considered procedurally correct.

2.6.3 Combining declarative and procedural views

The foregoing section has shown that the order of goals and clauses does matter. Furthermore, there are programs that are declaratively correct, but do not work in practice. Such discrepancies between the declarative and procedural meaning may

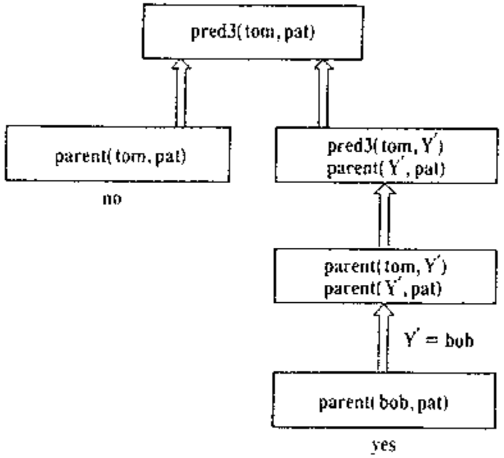
```
pred2( X, Z) :-
  parent( X, Y),
  pred2( Y, Z).

pred2( X, Z) :-
  parent( X, Z).
```



(a)

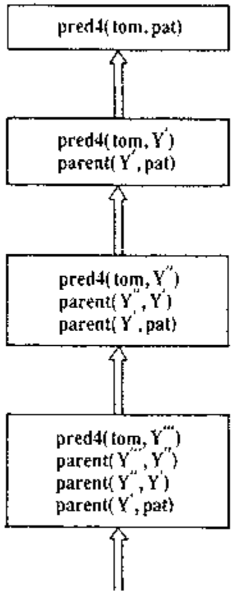
Figure 2.17 The behaviour of three formulations of the predecessor relation on the question: Is Tom a predecessor of Pat?

```
pred3( X, Z) :-
   parent( X, Z).
pred3( X, Z) :-
   pred3( X, Y),
   parent( Y, Z).
```



(b)

```
pred4( X, Z) :-
   pred4( X, Y),
   parent( Y, Z).
pred4( X, Z) :-
   parent( X, Z).
```



(c)

Figure 2.17 *contd*

appear annoying. One may argue: Why not simply forget about the declarative meaning? This argument can be brought to an extreme with a clause such as:

```
predecessor( X, Z) :- predecessor( X, Z).
```

which is declaratively correct, but is completely useless as a working program.

The reason why we should not forget about the declarative meaning is that progress in programming technology is achieved by moving away from procedural details toward declarative aspects, which are normally easier to formulate and understand. The system itself, not the programmer, should carry the burden of filling in the procedural details. Prolog does help toward this end, although, as we have seen in this section, it only helps partially: sometimes it does work out the procedural details itself properly, and sometimes it does not. The philosophy adopted by many is that it is better to have at least *some* declarative meaning rather than *none* ('none' is the case in most other programming languages). The practical aspect of this view is that it is often rather easy to get a working program once we have a program that is declaratively correct. Consequently, a useful practical approach that often works is to concentrate on the declarative aspects of the problem, then test the resulting program, and if it fails procedurally try to rearrange the clauses and goals into a suitable order.

2.7 The relation between Prolog and logic

Prolog is related to mathematical logic, so its syntax and meaning can be specified most concisely with references to logic. Prolog is indeed often defined that way. However, such an introduction to Prolog assumes that the reader is familiar with certain concepts of mathematical logic. These concepts are, on the other hand, certainly not necessary for understanding and using Prolog as a programming tool, which is the aim of this book. For the reader who is especially interested in the relation between Prolog and logic, the following are some basic links to mathematical logic, together with some appropriate references.

Prolog's syntax is that of the *first-order predicate logic* formulas written in the so-called *clause form* (a conjunctive normal form in which quantifiers are not explicitly written), and further restricted to Horn clauses only (clauses that have at most one positive literal). Clocksin and Mellish (1987) give a Prolog program that transforms a first-order predicate calculus formula into the clause form. The procedural meaning of Prolog is based on the *resolution principle* for mechanical theorem proving introduced by Robinson in his classic paper (1965). Prolog uses a special strategy for resolution theorem proving called SLD. An introduction to the first-order predicate calculus and resolution-based theorem proving can be found in several general books on artificial intelligence (Genesereth and Nilsson 1987; Ginsberg 1993; Poole *et al.* 1998; Russell and Norvig 1995; see also Flach 1994). Mathematical

questions regarding the properties of Prolog's procedural meaning with respect to logic are analyzed by Lloyd (1991).

Matching in Prolog corresponds to what is called *unification* in logic. However, we avoid the word unification because matching, for efficiency reasons in most Prolog systems, is implemented in a way that does not exactly correspond to unification (see Exercise 2.10). But from the practical point of view this approximation to unification is quite adequate. Proper unification requires the so-called *occurs check*: does a given variable occur in a given term? The occurs check would make matching inefficient.

Exercise

2.10 What happens if we ask Prolog:

```
?-   X = f( X).
```

Should this request for matching succeed or fail? According to the definition of unification in logic this should fail, but what happens according to our definition of matching in Section 2.2? Try to explain why many Prolog implementations answer the question above with:

```
X = f(f(f(f(f(f(f(f(f(f(f(f(f(f( ...
```

Summary

So far we have covered a kind of basic Prolog, also called 'pure Prolog'. It is 'pure' because it corresponds closely to formal logic. Extensions whose aim is to tailor the language toward some practical needs will be covered later in the book (Chapters 3, 5, 6, 7). Important points of this chapter are:

- Simple objects in Prolog are *atoms*, *variables* and *numbers*. Structured objects, or *structures*, are used to represent objects that have several components.
- Structures are constructed by means of *functors*. Each functor is defined by its name and arity.
- The type of object is recognized entirely by its syntactic form.
- The *lexical* scope of variables is one clause. Thus the same variable name in two clauses means two different variables.
- Structures can be naturally pictured as trees. Prolog can be viewed as a language for processing trees.
- The *matching* operation takes two terms and tries to make them identical by instantiating the variables in both terms.

- Matching, if it succeeds, results in the *most general* instantiation of variables.
- The *declarative semantics* of Prolog defines whether a goal is true with respect to a given program, and if it is true, for what instantiation of variables it is true.
- A comma between goals means the conjunction of goals. A semicolon between goals means the disjunction of goals.
- The *procedural semantics* of Prolog is a procedure for satisfying a list of goals in the context of a given program. The procedure outputs the truth or falsity of the goal list and the corresponding instantiations of variables. The procedure automatically backtracks to examine alternatives.
- The declarative meaning of programs in 'pure Prolog' does not depend on the order of clauses and the order of goals in clauses.
- The procedural meaning does depend on the order of goals and clauses. Thus the order can affect the efficiency of the program; an unsuitable order may even lead to infinite recursive calls.
- Given a declaratively correct program, changing the order of clauses and goals can improve the program's efficiency while retaining its declarative correctness. Reordering is one method of preventing indefinite looping.
- There are other more general techniques, apart from reordering, to prevent indefinite looping and thereby make programs procedurally robust.
- Concepts discussed in this chapter are:

  data objects: atom, number, variable, structure
  term
  functor, arity of a functor
  principal functor of a term
  matching of terms
  most general instantiation
  declarative semantics
  instance of a clause, variant of a clause
  procedural semantics
  executing goals

References

Clocksin, W.F. and Mellish, C.S. (1987) *Programming in Prolog*, second edition. Berlin: Springer-Verlag.

Flach, P. (1994) *Simply Logical: Intelligent Reasoning by Example*. Chichester, UK: Wiley.

Genesereth, M.R. and Nilsson, N.J. (1987) *Logical Foundation of Artificial Intelligence*. Palo Alto, CA: Morgan Kaufmann.

Ginsberg, M. (1993) *Essentials of Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann.

Lloyd, J.W. (1991) *Foundations of Logic Programming*, second edition. Berlin: Springer-Verlag.

Robinson, A.J. (1965) A machine-oriented logic based on the resolution principle. *JACM* 12: 23–41.

Poole, D., Mackworth, A. and Gaebel, R. (1998) *Computational Intelligence: A Logical Approach.* Oxford University Press.

Russell, S. and Norvig, P. (1995) *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

chapter 3

Lists, Operators, Arithmetic

In this chapter we will study a special notation for lists, one of the simplest and most useful structures, and some programs for typical operations on lists. We will also look at simple arithmetic and the operator notation, which often improves the readability of programs. Basic Prolog of Chapter 2, extended with these three additions, becomes a convenient framework for writing interesting programs.

3.1 Representation of lists

The *list* is a simple data structure widely used in non-numeric programming. A list is a sequence of any number of items, such as ann, tennis, tom, skiing. Such a list can be written in Prolog as:

```
[ ann, tennis, tom, skiing]
```

This is, however, only the external appearance of lists. As we have already seen in Chapter 2, all structured objects in Prolog are trees. Lists are no exception to this.

How can a list be represented as a standard Prolog object? We have to consider two cases: the list is either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list can be viewed as consisting of two things:

(1) the first item, called the *head* of the list;

(2) the remaining part of the list, called the *tail.*

For our example list,

```
[ ann, tennis, tom, skiing]
```

the head is ann and the tail is the list:

```
[ tennis, tom, skiing]
```

In general, the head can be anything (any Prolog object, for example, a tree or a variable); the tail has to be a list. The head and the tail are then combined into a structure by a special functor,

```
.( Head, Tail)
```

Since Tail is in turn a list, it is either empty or it has its own head and tail. Therefore, to represent lists of any length no additional principle is needed. Our example list is then represented as the term:

```
.( ann, .( tennis, .( tom, .( skiing, [ ] ) ) ) )
```

Figure 3.1 shows the corresponding tree structure. Note that the empty list appears in our term. This is because the one but last tail is a single item list:

```
[ skiing]
```

This list has the empty list as its tail:

```
[ skiing] = .( skiing, [ ] )
```

This example shows how the general principle for structuring data objects in Prolog also applies to lists of any length. As our example also shows, the straightforward notation with dots and possibly deep nesting of subterms in the tail part can produce rather confusing expressions. This is the reason why Prolog provides the neater notation for lists, so that they can be written as sequences of items enclosed in square brackets. A programmer can use both notations, but the square bracket notation is, of course, normally preferred. We will be aware, however, that this is only a cosmetic improvement and that our lists will be internally represented as binary trees. When such terms are output they will be automatically converted into their neater form. Thus the following conversation with Prolog is possible:

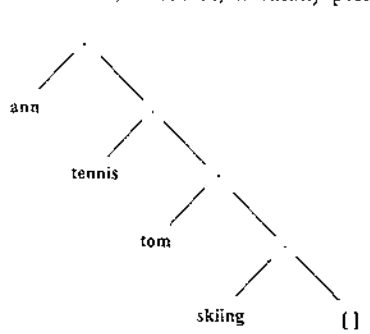


Figure 3.1 Tree representation of the list [ann, tennis, tom, skiing].


```
?- List1 = [a,b,c],
   List2 = .( a, .( b, .( c, [ ] ) ) ).

List1 = [a,b,c]
List2 = [a,b,c]

?- Hobbies1 = .( tennis, .( music, [ ] ) ),
   Hobbies2 = [ skiing, food],
   L = [ ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [ tennis, music]
Hobbies2 = [ skiing, food]
L = [ ann, [tennis,music], tom, [skiing,food] ]
```

This example also reminds us that the elements of a list can be objects of any kind; in particular they can also be lists.

It is often practical to treat the whole tail as a single object. For example, let:

```
L = [a,b,c]
```

Then we could write:

```
Tail = [b,c]   and   L = .( a, Tail)
```

To express this in the square bracket notation for lists, Prolog provides another notational extension, the vertical bar, which separates the head and the tail:

```
L = [ a | Tail]
```

The vertical bar notation is in fact more general: we can list any number of elements followed by '|' and the list of remaining items. Thus alternative ways of writing the above list are:

```
[a,b,c] = [a | [b,c] ] = [a,b | [c] ] = [a,b,c | [ ] ]
```

To summarize:

- A list is a data structure that is either empty or consists of two parts: a *head* and a *tail*. The tail itself has to be a list.
- Lists are handled in Prolog as a special case of binary trees. For improved readability Prolog provides a special notation for lists, thus accepting lists written as:

  ```
  [ Item1, Item2, ... ]
  ```

  or

  ```
  [ Head | Tail]
  ```

  or

  ```
  [ Item1, Item2, ... | Others]
  ```

3.2 Some operations on lists

Lists can be used to represent sets, although there is a difference: the order of elements in a set does not matter while the order of items in a list does; also, the same object can occur repeatedly in a list. Still, the most common operations on lists are similar to those on sets. Among them are:

- checking whether some object is an element of a list, which corresponds to checking for the set membership;
- concatenation of two lists, obtaining a third list, which may correspond to the union of sets;
- adding a new object to a list, or deleting some object from it.

In the remainder of this section we give programs for these and some other operations on lists.

3.2.1 Membership

Let us implement the membership relation as:

```
member( X, L)
```

where X is an object and L is a list. The goal member(X, L) is true if X occurs in L. For example,

```
member( b, [a,b,c] )
```

is true,

```
member( b, [a,[b,c]] )
```

is not true, but

```
member( [b,c], [a,[b,c]] )
```

is true. The program for the membership relation can be based on the following observation:

X is a member of L if either:
(1) X is the head of L, or
(2) X is a member of the tail of L.

This can be written in two clauses; the first is a simple fact and the second is a rule:

```
member( X, [X | Tail] ).

member( X, [Head | Tail] ) :-
  member( X, Tail).
```


3.2.2 Concatenation

For concatenating lists we will define the relation:

```
conc( L1, L2, L3)
```

Here L1 and L2 are two lists, and L3 is their concatenation. For example,

```
conc( [a,b], [c,d], [a,b,c,d] )
```

is true, but

```
conc( [a,b], [c,d], [a,b,a,c,d] )
```

is false. In the definition of conc we will have again two cases, depending on the first argument, L1:

(1) If the first argument is the empty list then the second and the third arguments must be the same list (call it L); this is expressed by the following Prolog fact:

```
conc( [ ], L, L).
```

(2) If the first argument of conc is a non-empty list then it has a head and a tail and must look like this:

```
[X | L1]
```

Figure 3.2 illustrates the concatenation of [X | L1] and some list L2. The result of the concatenation is the list [X | L3] where L3 is the concatenation of L1 and L2. In Prolog this is written as:

```
conc( [X | L1], L2, [X | L3] ) :-
  conc( L1, L2, L3).
```

This program can now be used for concatenating given lists, for example:

```
?- conc( [a,b,c], [1,2,3], L).
L = [a,b,c,1,2,3]
```

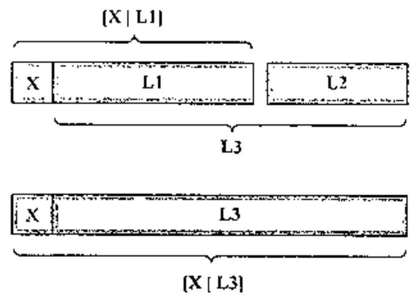


Figure 3.2 Concatenation of lists.

```
?- conc( [a,[b,c],d], [a,[ ],b], L).

L = [a, [b,c], d, a, [ ], b]
```

Although the conc program looks rather simple it can be used flexibly in many other ways. For example, we can use conc in the inverse direction for *decomposing* a given list into two lists, as follows:

```
?- conc( L1, L2, [a,b,c] ).

L1 = [ ]
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [ ];

no
```

It is possible to decompose the list [a,b,c] in four ways, all of which were found by our program through backtracking.

We can also use our program to look for a certain pattern in a list. For example, we can find the months that precede and the months that follow a given month, as in the following goal:

```
?- conc( Before, [may | After],
        [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

Further we can find the immediate predecessor and the immediate successor of May by asking:

```
?- conc( _, [Month1,may,Month2 | _],
        [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Month1 = apr
Month2 = jun
```

Further still, we can, for example, delete from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's. For example:

```
?- L1 = [a,b,z,z,c,z,z,z,d,e],
conc( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

We have already programmed the membership relation. Using conc, however, the membership relation could be elegantly programmed by the clause:


```
member1( X, L) :-
  conc( L1, [X | L2], L).
```

This clause says: X is a member of list L if L can be decomposed into two lists so that the second one has X as its head. Of course, member1 defines the same relation as member. We have just used a different name to distinguish between the two implementations. Note that the above clause can be written using anonymous variables as:

```
member1( X, L) :-
  conc( _, [X | _], L).
```

It is interesting to compare both implementations of the membership relation, member and member1. member has a rather straightforward procedural meaning, which is as follows:

To check whether some X is a member of some list L:
(1) first check whether the head of L is equal to X, and then
(2) check whether X is a member of the tail of L.

On the other hand, the declarative reading of member1 is straightforward, but its procedural meaning is not so obvious. An interesting exercise is to find how member1 actually computes something. An example execution trace will give some idea: let us consider the question:

```
?- member1( b, [a,b,c] ).
```

Figure 3.3 shows the execution trace. From the trace we can infer that member1 behaves similarly to member. It scans the list, element by element, until the item in question is found or the list is exhausted.

Exercises

3.1 (a) Write a goal, using conc, to delete the last three elements from a list L producing another list L1. Hint: L is the concatenation of L1 and a three-element list.

(b) Write a goal to delete the first three elements and the last three elements from a list L producing list L2.

3.2 Define the relation

```
last( Item, List)
```

so that Item is the last element of a list List. Write two versions: (a) using the conc relation, (b) without conc.

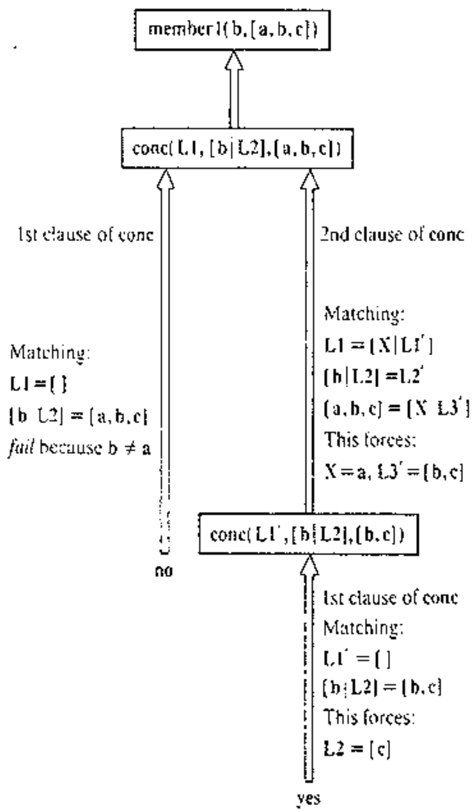


Figure 3.3 Procedure **member1** finds an item in a given list by sequentially searching the list.

3.2.3 Adding an item

To add an item to a list, it is easiest to put the new item in front of the list so that it becomes the new head. If X is the new item and the list to which X is added is L then the resulting list is simply:

```
[X | L]
```

So we actually need no procedure for adding a new element in front of the list. Nevertheless, if we want to define such a procedure explicitly, it can be written as the fact:

```
add( X, L, [X | L] ).
```


3.2.4 Deleting an item

Deleting an item, X, from a list, L, can be programmed as a relation

```
del( X, L, L1)
```

where L1 is equal to the list L with the item X removed. The del relation can be defined similarly to the membership relation. We have, again, two cases:

(1) If X is the head of the list then the result after the deletion is the tail of the list.

(2) If X is in the tail then it is deleted from there.

```
del( X, [X | Tail], Tail).

del( X, [Y | Tail], [Y | Tail1] ) :-
  del( X, Tail, Tail1).
```

Like **member**, **del** is also non-deterministic. If there are several occurrences of X in the list then del will be able to delete any one of them by backtracking. Of course, each alternative execution will only delete one occurrence of X, leaving the others untouched. For example:

```
?- del( a, [a,b,a,a], L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```

del will fail if the list does not contain the item to be deleted.

del can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list. For example, if we want to insert a at any place in the list [1,2,3] then we can do this by asking the question: What is L such that after deleting a from L we obtain [1,2,3]?

```
?- del( a, L, [1,2,3] ).
L = [a,1,2,3];
L = [1,a,2,3];
L = [1,2,a,3];
L = [1,2,3,a];
no
```

In general, the operation of inserting X at any place in some list List giving BiggerList can be defined by the clause:

```
insert( X, List, BiggerList) :-
  del( X, BiggerList, List).
```

In member1 we elegantly implemented the membership relation by using conc. We can also use del to test for membership. The idea is simple: some X is a member of List if X can be deleted from List:

```
member2( X, List) :-
  del( X, List, _).
```

3.2.5 Sublist

Let us now consider the sublist relation. This relation has two arguments, a list L and a list S such that S occurs within L as its sublist. So,

```
sublist( [c,d,e], [a,b,c,d,e,f] )
```

is true, but

```
sublist( [c,e], [a,b,c,d,e,f] )
```

is not. The Prolog program for sublist can be based on the same idea as member1, only this time the relation is more general (see Figure 3.4). Accordingly, the relation can be formulated as:

S is a sublist of L if:
(1) L can be decomposed into two lists, L1 and L2, and
(2) L2 can be decomposed into two lists, S and some L3.

As we have seen before, the conc relation can be used for decomposing lists. So the above formulation can be expressed in Prolog as:

```
sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).
```

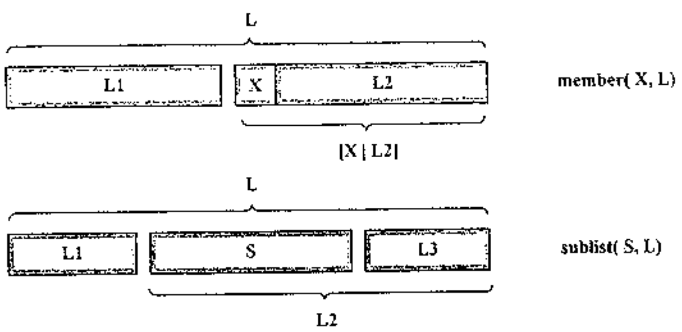


Figure 3.4 The member and sublist relations.

Of course, the sublist procedure can be used flexibly in several ways. Although it was designed to check if some list occurs as a sublist within another list it can also be used, for example, to find all sublists of a given list:

```
?- sublist( S, [a,b,c] ).
S = [];
S = [a];
S = [a,b];
S = [a,b,c];
S = [];
S = [b];
...
```

3.2.6 Permutations

Sometimes it is useful to generate permutations of a given list. To this end, we will define the permutation relation with two arguments. The arguments are two lists such that one is a permutation of the other. The intention is to generate permutations of a list through backtracking using the permutation procedure, as in the following example:

```
?- permutation( [a,b,c], P).
P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
...
```

The program for permutation can be, again, based on the consideration of two cases, depending on the first list:

(1) If the first list is empty then the second list must also be empty.
(2) If the first list is not empty then it has the form [X | L], and a permutation of such a list can be constructed as shown in Figure 3.5: first permute L obtaining L1 and then insert X at any position into L1.

Two Prolog clauses that correspond to these two cases are:

```
permutation( [], [] ).

permutation( [X | L], P) :-
  permutation( L, L1),
  insert( X, L1, P).
```

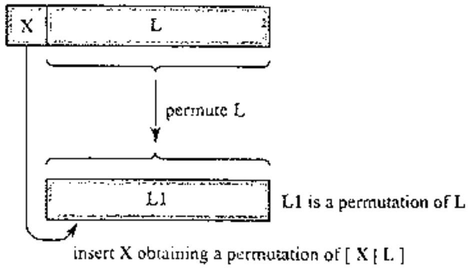


Figure 3.5 One way of constructing a permutation of the list [X | L].

One alternative to this program would be to delete an element, X, from the first list, permute the rest of it obtaining a list P, and then add X in front of P. The corresponding program is:

```
permutation2( [], [] ).

permutation2( L, [X | P] ) :-
  del( X, L, L1),
  permutation2( L1, P).
```

It is instructive to do some experiments with our permutation programs. Its normal use would be something like this:

```
?- permutation( [red,blue,green], P).
```

This would result in all six permutations, as intended:

```
P = [ red, blue, green];
P = [ red, green, blue];
P = [ blue, red, green];
P = [ blue, green, red];
P = [ green, red, blue];
P = [ green, blue, red];
no
```

Another attempt to use **permutation** is:

```
?- permutation( L, [a,b,c] ).
```

Our first version, **permutation**, will now instantiate L successfully to all six permutations. If the user then requests more solutions, the program would never answer 'no' because it would get into an infinite loop trying to find another permutation when there is none. Our second version, **permutation2**, will in this case find only the first (identical) permutation and then immediately get into an infinite loop. Thus, some care is necessary when using these permutation programs.

Exercises

3.3 Define two predicates

evenlength(List) and **oddlength(List)**

so that they are true if their argument is a list of even or odd length respectively. For example, the list [a,b,c,d] is 'evenlength' and [a,b,c] is 'oddlength'.

3.4 Define the relation

reverse(List, ReversedList)

that reverses lists. For example, reverse([a,b,c,d], [d,c,b,a]).

3.5 Define the predicate **palindrome(List)**. A list is a palindrome if it reads the same in the forward and in the backward direction. For example, [m,a,d,a,m].

3.6 Define the relation

shift(List1, List2)

so that List2 is List1 'shifted rotationally' by one element to the left. For example,

```
?- shift( [1,2,3,4,5], L1),
   shift( L1, L2).
```

produces:

```
L1 = [2,3,4,5,1]
L2 = [3,4,5,1,2]
```

3.7 Define the relation

translate(List1, List2)

to translate a list of numbers between 0 and 9 to a list of the corresponding words. For example:

translate([3,5,1,3], [three,five,one,three])

Use the following as an auxiliary relation:

means(0, zero). means(1, one). means(2, two). ...

3.8 Define the relation

subset(Set, Subset)

where Set and Subset are two lists representing two sets. We would like to be able to use this relation not only to check for the subset relation, but also to generate all possible subsets of a given set. For example:

```
?- subset( [a,b,c], S).
S = [a,b,c];
S = [a,b];
S = [a,c];
S = [a];
S = [b,c];
S = [b];
...
```

3.9 Define the relation

```
dividelist( List, List1, List2)
```

so that the elements of List are partitioned between List1 and List2, and List1 and List2 are of approximately the same length. For example, dividelist([a,b,c,d,e], [a,c,e], [b,d]).

3.10 Rewrite the monkey and banana program of Chapter 2 as the relation

```
canget( State, Actions)
```

to answer not just 'yes' or 'no', but to produce a sequence of monkey's actions represented as a list of moves. For example:

```
Actions = [ walk(door,window), push(window,middle), climb, grasp]
```

3.11 Define the relation

```
flatten( List, FlatList)
```

where List can be a list of lists, and FlatList is List 'flattened' so that the elements of List's sublists (or sub-sublists) are reorganized as one plain list. For example:

```
?- flatten( [a,b,[c,d],[ ],[[[e]]],f], L).
L = [a,b,c,d,e,f]
```

3.3 Operator notation

In mathematics we are used to writing expressions like

$2*a + b*c$

where + and * are operators, and 2, a, b, are arguments. In particular, + and * are said to be *infix* operators because they appear *between* the two arguments. Such expressions can be represented as trees, as in Figure 3.6, and can be written as Prolog terms with + and * as functors:

```
+( *(2,a), *(b,c) )
```

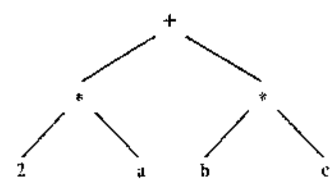



Figure 3.6 Tree representation of the expression 2*a + b*c.

Since we would normally prefer to have such expressions written in the usual, infix style with operators, Prolog caters for this notational convenience. Prolog will therefore accept our expression written simply as:

```
2*a + b*c
```

This will be, however, only the external representation of this object, which will be automatically converted into the usual form of Prolog terms. Such a term will be output for the user, again, in its external, infix form.

Thus operators in Prolog are merely a notational extension. If we write a + b, Prolog will handle it exactly as if it had been written +(a,b). In order that Prolog properly understands expressions such as a – b*c, Prolog has to know that * binds stronger than +. We say that + has higher precedence than *. So the precedence of operators decides what is the correct interpretation of expressions. For example, the expression a + b*c can be, in principle, understood either as

```
+( a, *(b,c) )
```

or as

```
*( +(a,b), c)
```

The general rule is that the operator with the highest precedence is the principal functor of the term. If expressions containing + and * are to be understood according to our normal conventions, then + has to have a higher precedence than *. Then the expression a + b*c means the same as a + (b*c). If another interpretation is intended, then it has to be explicitly indicated by parentheses – for example, (a + b)*c.

A programmer can define his or her own operators. So, for example, we can define the atoms **has** and **supports** as infix operators and then write in the program facts like:

```
peter has information.
floor supports table.
```

These facts are exactly equivalent to:

```
has( peter, information).
supports( floor, table).
```

A programmer can define new operators by inserting into the program special kinds of clauses, sometimes called *directives*, which act as operator definitions. An operator definition must appear in the program before any expression containing that operator. For our example, the operator has can be properly defined by the directive:

```
:- op( 600, xfx, has).
```

This tells Prolog that we want to use 'has' as an operator, whose precedence is 600 and its type is 'xfx', which is a kind of infix operator. The form of the specifier 'xfx' suggests that the operator, denoted by 'f', is between the two arguments denoted by 'x'.

Notice that operator definitions do not specify any operation or action. In principle, *no operation on data is associated with an operator* (except in very special cases). Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data, although the word 'operator' appears to suggest an action.

Operator names are atoms. An operator's precedence must be in some range which depends on the implementation. We will assume that the range is between 1 and 1200.

There are three groups of operator types which are indicated by type specifiers such as xfx. The three groups are:

(1) infix operators of three types:

    xfx xfy yfx

(2) prefix operators of two types:

    fx fy

(3) postfix operators of two types:

    xf yf

The specifiers are chosen so as to reflect the structure of the expression where 'f' represents the operator and 'x' and 'y' represent arguments. An 'f' appearing between the arguments indicates that the operator is infix. The prefix and postfix specifiers have only one argument, which follows or precedes the operator respectively.

There is a difference between 'x' and 'y'. To explain this we need to introduce the notion of the *precedence of argument*. If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0; if an argument is a structure then its precedence is equal to the precedence of its principal functor. 'x' represents an argument whose precedence must be strictly lower than that of the operator. 'y' represents an argument whose precedence is lower or equal to that of the operator.

These rules help to disambiguate expressions with several operators of the same precedence. For example, the expression

```
a - b - c
```

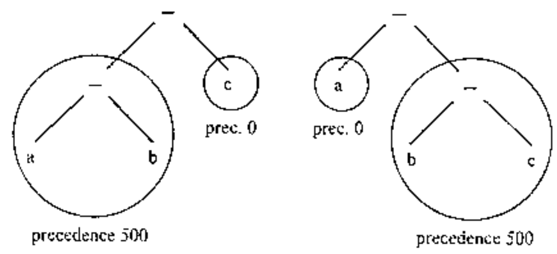



Figure 3.7 Two interpretations of the expression a – b – c assuming that '–' has precedence 500. If '–' is of type yfx, then interpretation 2 is invalid because the precedence of b – c is not less than the precedence of '–'.

is normally understood as (a – b) – c, and not as a – (b – c). To achieve the normal interpretation the operator '–' has to be defined as yfx. Figure 3.7 shows why the second interpretation is then ruled out.

As another example consider the prefix operator not. If not is defined as fy then the expression

```
not not p
```

is legal; but if not is defined as fx then this expression is illegal because the argument to the first not is not p, which has the same precedence as not itself. In this case the expression has to be written with parentheses:

```
not( not p)
```

For convenience, some operators are predefined in the Prolog system so that they can be readily used, and no definition is needed for them. What these operators are and what their precedences are depends on the implementation of Prolog. We will assume that this set of 'standard' operators is as if defined by the clauses in Figure 3.8. The operators in this figure are a subset of those defined in the Prolog standard, plus the operator not. As Figure 3.8 also shows, several operators can be declared by one clause if they all have the same precedence and if they are all of the same type. In this case the operators' names are written as a list.

The use of operators can greatly improve the readability of programs. As an example let us assume that we are writing a program for manipulating Boolean expressions. In such a program we may want to state, for example, one of de Morgan's equivalence theorems, which can in mathematics be written as:

~(A & B) <===> ~A v ~B

One way to state this in Prolog is by the clause:

```
equivalence( not( and( A, B)), or( not( A), not( B) ) ).
```

```
:- op( 1200, xfx, [ :-, --> ] ).
:- op( 1200, fx [ :-, ?-] ).
:- op( 1100, xfy, ';' ).
:- op( 1050, xfy, -> ).
:- op( 1000, xfy, ',' ).
:- op( 900, fy, [ not, '\+'] ).
:- op( 700, xfx, [ =, \=, ==, \==, =.. ] ).
:- op( 700, xfx, [ is, =:=, =\=, < , =< , > , >=, @<, @=<, @>, @>=] ).
:- op( 500, yfx, [ + , - ] ).
:- op( 400, yfx, [ *, /, //, mod] ).
:- op( 200, xfx, **).
:- op( 200, xfy, ^).
:- op( 200, fy, - ).
```

Figure 3.8 A set of predefined operators.

However, it is in general a good programming practice to try to retain as much resemblance as possible between the original problem notation and the notation used in the program. In our example, this can be achieved almost completely by using operators. A suitable set of operators for our purpose can be defined as:

```
:- op( 800, xfx, <===>).
:- op( 700, xfy, v).
:- op( 600, xfy, &).
:- op( 500, fy, ~).
```

Now the de Morgan's theorem can be written as the fact:

```
~(A & B) <===> ~A v ~B.
```

According to our specification of operators above, this term is understood as shown in Figure 3.9.

To summarize:

- The readability of programs can be often improved by using the operator notation. Operators can be infix, prefix or postfix.
- In principle, no operation on data is associated with an operator except in special cases. Operator definitions do not define any action, they only introduce new notation. Operators, as functors, only hold together components of structures.
- A programmer can define his or her own operators. Each operator is defined by its name, precedence and type.
- The precedence is an integer within some range, usually between 1 and 1200. The operator with the highest precedence in the expression is the principal functor of the expression. Operators with lowest precedence bind strongest.

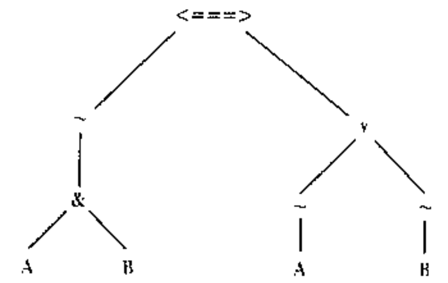


Figure 3.9 Interpretation of the term ~(A & B) <===> ~A v ~B.

- The type of an operator depends on two things: (1) the position of the operator with respect to the arguments, and (2) the precedence of the arguments compared to the precedence of the operator itself. In a specifier like xfy, x indicates an argument whose precedence is strictly lower than that of the operator; y indicates an argument whose precedence is less than or equal to that of the operator.

Exercises

3.12 Assuming the operator definitions

```
:- op( 300, xfx, plays).
:- op( 200, xfy, and).
```

then the following two terms are syntactically legal objects:

```
Term1 = jimmy plays football and squash
Term2 = susan plays tennis and basketball and volleyball
```

How are these terms understood by Prolog? What are their principal functors and what is their structure?

3.13 Suggest an appropriate definition of operators ('was', 'of', 'the') to be able to write clauses like

```
diana was the secretary of the department.
```

and then ask Prolog:

```
?- Who was the secretary of the department.
Who = diana

?- diana was What.
What = the secretary of the department
```

3.14 Consider the program:

```
t( 0+1, 1+0).
t( X+0+1, X+1+0).
t( X+1+1, Z) :-
  t( X+1, X1),
  t( X1+1, Z).
```

How will this program answer the following questions if '+' is an infix operator of type yfx (as usual):

(a) ?- t(0+1, A).

(b) ?- t(0+1+1, B).

(c) ?- t(1+0+1+1+1, C).

(d) ?- t(D, 1+1+1+0).

3.15 In the previous section, relations involving lists were written as:

```
member( Element, List),
conc( List1, List2, List3),
del( Element, List, NewList), ...
```

Suppose that we would prefer to write these relations as:

```
Element in List,
concatenating List1 and List2 gives List3,
deleting Element from List gives NewList, ...
```

Define 'in', 'concatenating', 'and', etc. as operators to make this possible. Also, redefine the corresponding procedures.

3.4 Arithmetic

Some of the predefined operators can be used for basic arithmetic operations. These are:

+	addition
-	subtraction
*	multiplication
/	division
**	power
//	integer division
mod	modulo, the remainder of integer division

Notice that this is an exceptional case in which an operator may in fact invoke an operation. But even in such cases an additional indication to perform arithmetic

will be necessary. The following question is a naive attempt to request arithmetic computation:

```
?- X = 1 + 2.
```

Prolog will 'quietly' answer

```
X = 1 + 2
```

and not X = 3 as we might possibly expect. The reason is simple: the expression 1 + 2 merely denotes a Prolog term where + is the functor and 1 and 2 are its arguments. There is nothing in the above goal to force Prolog to actually activate the addition operation. A special predefined operator, is, is provided to circumvent this problem. The is operator will force evaluation. So the right way to invoke arithmetic is:

```
?- X is 1 + 2.
```

Now the answer will be:

```
X = 3
```

The addition here was carried out by a special procedure that is associated with the operator is. We call such procedures *built-in procedures*.

Different implementations of Prolog may use somewhat different notations for arithmetics. For example, the '/' operator may denote integer division or real division. In this book, '/' denotes real division, the operator // denotes integer division, and mod denotes the remainder. Accordingly, the question:

```
?- X is 5/2,
   Y is 5//2,
   Z is 5 mod 2.
```

is answered by:

```
X = 2.5
Y = 2
Z = 1
```

The left argument of the is operator is a simple object. The right argument is an arithmetic expression composed of arithmetic operators, numbers and variables. Since the is operator will force the evaluation, all the variables in the expression must already be instantiated to numbers at the time of execution of this goal. The precedence of the predefined arithmetic operators (see Figure 3.8) is such that the associativity of arguments with operators is the same as normally in mathematics. Parentheses can be used to indicate different associations. Note that +, -, *, / and div are defined as yfx, which means that evaluation is carried out from left to right. For example,

```
X is 5 - 2 - 1
```

is interpreted as:

```
X is (5 - 2) - 1
```

Prolog implementations usually also provide standard functions such as sin(X), cos(X), atan(X), log(X), exp(X), etc. These functions can appear to the right of operator is.

Arithmetic is also involved when *comparing* numerical values. We can, for example, test whether the product of 277 and 37 is greater than 10000 by the goal:

```
?- 277 * 37 > 10000.

yes
```

Note that, similarly to is, the '>' operator also forces the evaluation.

Suppose that we have in the program a relation born that relates the names of people with their birth years. Then we can retrieve the names of people born between 1980 and 1990 inclusive with the following question:

```
?- born( Name, Year),
   Year >= 1980,
   Year =< 1990.
```

The comparison operators are as follows:

```
X > Y      X is greater than Y
X < Y      X is less than Y
X >= Y     X is greater than or equal to Y
X =< Y     X is less than or equal to Y
X =:= Y    the values of X and Y are equal
X =\= Y    the values of X and Y are not equal
```

Notice the difference between the matching operator '=' and '=:='; for example, in the goals X = Y and X =:= Y. The first goal will cause the matching of the objects X and Y, and will, if X and Y match, possibly instantiate some variables in X and Y. There will be no evaluation. On the other hand, X =:= Y causes the arithmetic evaluation and cannot cause any instantiation of variables. These differences are illustrated by the following examples:

```
?- 1 + 2 =:= 2 + 1.

yes

?- 1 + 2 = 2 + 1.

no

?- 1 + A = B + 2.

A = 2
B = 1
```


Let us further illustrate the use of arithmetic operations by two simple examples. The first is computing the greatest common divisor; the second, counting the items in a list.

Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:

(1) If X and Y are equal then D is equal to X.

(2) If X < Y then D is equal to the greatest common divisor of X and the difference Y – X.

(3) If Y < X then do the same as in case (2) with X and Y interchanged.

It can be easily shown by an example that these three rules actually work. Choosing, for example, X = 20 and Y = 25, the above rules would give D = 5 after a sequence of subtractions.

These rules can be formulated into a Prolog program by defining a three-argument relation, say:

```
gcd( X, Y, D)
```

The three rules are then expressed as three clauses, as follows:

```
gcd( X, X, X).

gcd( X, Y, D) :-
  X < Y,
  Y1 is Y - X,
  gcd( X, Y1, D).

gcd( X, Y, D) :-
  Y < X,
  gcd( Y, X, D).
```

Of course, the last goal in the third clause could be equivalently replaced by the two goals:

```
X1 is X - Y,
gcd( X1, Y, D)
```

Our next example involves counting, which usually requires some arithmetic. An example of such a task is to establish the length of a list; that is, we have to count the items in the list. Let us define the procedure:

```
length( List, N)
```

which will count the elements in a list List and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases:

(1) If the list is empty then its length is 0.

(2) If the list is not empty then List = [Head | Tail]; then its length is equal to 1 plus the length of the tail Tail.

These two cases correspond to the following program:

```
length( [], 0).

length( [_ | Tail], N) :-
  length( Tail, N1),
  N is 1 + N1.
```

An application of length can be:

```
?- length( [a,b,[c,d],e], N).

N = 4
```

Note that in the second clause of **length**, the two goals of the body cannot be swapped. The reason for this is that N1 has to be instantiated before the goal:

```
N is 1 + N1
```

can be processed. With the built-in procedure is, a relation has been introduced that is sensitive to the order of processing and therefore the procedural considerations have become vital.

It is interesting to see what happens if we try to program the length relation without the use of is. Such an attempt can be:

```
length1( [], 0).

length1( [_ | Tail], N) :-
  length1( Tail, N1),
  N = 1 + N1.
```

Now the goal

```
?- length1( [a,b,[c,d],e], N).
```

will produce the answer:

```
N = 1+(1+(1+(1+0))).
```

The addition was never explicitly forced and was therefore not carried out at all. But in length1 we can, unlike in **length**, swap the goals in the second clause:

```
length1( [_ | Tail], N) :-
  N = 1 + N1,
  length1( Tail, N1).
```

This version of **length1** will produce the same result as the original version. It can also be written shorter, as follows,

```
length1( [_ | Tail], 1 + N) :-
  length1( Tail, N).
```

still producing the same result. We can, however, use **length1** to find the number of elements in a list as follows:


```
?- length1( [a,b,c], N), Length is N.

N = 1+(1+(1+0))
Length = 3
```

Finally we note that the predicate **length** is often provided as a built-in predicate.

To summarize:

- Built-in procedures can be used for doing arithmetic.
- Arithmetic operations have to be explicitly requested by the built-in procedure is. There are built-in procedures associated with the predefined operators +, −, *, /, div and mod.
- At the time that evaluation is carried out, all arguments must be already instantiated to numbers.
- The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

Exercises

3.16 Define the relation

```
max( X, Y, Max)
```

so that Max is the greater of two numbers X and Y.

3.17 Define the predicate

```
maxlist( List, Max)
```

so that Max is the greatest number in the list of numbers List.

3.18 Define the predicate

```
sumlist( List, Sum)
```

so that Sum is the sum of a given list of numbers List.

3.19 Define the predicate

```
ordered( List)
```

which is true if List is an ordered list of numbers. For example,

```
ordered( [1,5,6,6,9,12] ).
```

3.20 Define the predicate

```
subsum( Set, Sum, SubSet)
```

so that Set is a list of numbers, SubSet is a subset of these numbers, and the sum of the numbers in SubSet is Sum. For example:

```
?- subsum( [1,2,5,3,2], 5, Sub).
Sub = [1,2,2];
Sub = [2,3];
Sub = [5];
...
```

3.21 Define the procedure

```
between( N1, N2, X)
```

which, for two given integers N1 and N2, generates through backtracking all the integers X that satisfy the constraint N1 ≤ X ≤ N2.

3.22 Define the operators 'if', 'then', 'else' and ':=' so that the following becomes a legal term:

```
if X > Y then Z := X else Z := Y
```

Choose the precedences so that 'if' will be the principal functor. Then define the relation 'if' as a small interpreter for a kind of 'if-then-else' statement of the form

```
if Val1 > Val2 then Var := Val3 else Var := Val4
```

where Val1, Val2, Val3 and Val4 are numbers (or variables instantiated to numbers) and Var is a variable. The meaning of the 'if' relation should be: if the value of Val1 is greater than the value of Val2 then Var is instantiated to Val3, otherwise to Val4. Here is an example of the use of this interpreter:

```
?- X = 2, Y = 3,
   Val2 is 2*X,
   Val4 is 4*X,
   if Y > Val2 then Z := Y else Z := Val4,
   if Z > 5 then W := 1 else W := 0.

X = 2
Y = 3
Z = 8
W = 1
Val2 = 4
Val4 = 8
```

Summary

- The list is a frequently used structure. It is either empty or consists of a *head* and a *tail* which is a list as well. Prolog provides a special notation for lists.
- Common operations on lists, programmed in this chapter, are: list membership, concatenation, adding an item, deleting an item, sublist.

- The *operator notation* allows the programmer to tailor the syntax of programs toward particular needs. Using operators the readability of programs can be greatly improved.
- New operators are defined by the directive op, stating the name of an operator, its type and precedence.
- In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.
- Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure is and by the comparison predicates <, =<, etc.
- Concepts introduced in this chapter are:

  list, head of list, tail of list
  list notation
  operators, operator notation
  infix, prefix and suffix operators
  precedence of an operator
  arithmetic built-in procedures

chapter 4

Using Structures: Example Programs

Data structures, with matching, backtracking and arithmetic, are a powerful programming tool. In this chapter we will develop the skill of using this tool through programming examples: retrieving structured information from a database, simulating a non-deterministic automaton, travel planning, and eight queens on the chessboard. We will also see how the principle of data abstraction can be carried out in Prolog. The programming examples in this chapter can be read selectively.

4.1 Retrieving structured information from a database

This exercise develops techniques of representing and manipulating structured data objects. It also illustrates Prolog as a natural database query language.

A database can be naturally represented in Prolog as a set of facts. For example, a database about families can be represented so that each family is described by one clause. Figure 4.1 shows how the information about each family can be structured. Each family has three components: husband, wife and children. As the number of children varies from family to family the children are represented by a list that is capable of accommodating any number of items. Each person is, in turn, represented by a structure of four components: name, surname, date of birth, job. The job information is 'unemployed', or it specifies the working organization and salary. The family of Figure 4.1 can be stored in the database by the clause:

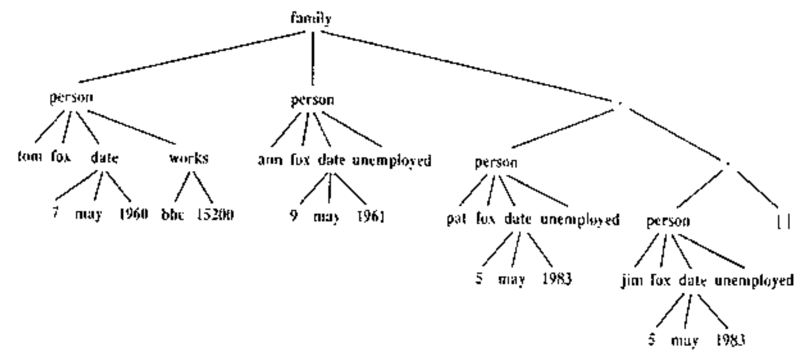


Figure 4.1 Structuring information about the family.

```
family(
  person( tom, fox, date(7,may,1960), works(bbc,15200) ),
  person( ann, fox, date(9,may,1961), unemployed),
  [ person( pat, fox, date(5,may,1983), unemployed),
    person( jim, fox, date(5,may,1983), unemployed) ] ).
```

Our database would then be comprised of a sequence of facts like this describing all families that are of interest to our program.

Prolog is, in fact, a very suitable language for retrieving the desired information from such a database. One nice thing about Prolog is that we can refer to objects without actually specifying all the components of these objects. We can merely indicate the *structure* of objects that we are interested in, and leave the particular components in the structures unspecified or only partially specified. Figure 4.2 shows some examples. So we can refer to all Armstrong families by:

```
family( person( _, armstrong, _, _), _, _)
```

The underscore characters denote different anonymous variables; we do not care about their values. Further, we can refer to all families with three children by the term:

```
family( _, _, [_, _, _] )
```

To find all married women that have at least three children we can pose the question:

```
?- family( _, person( Name, Surname, _, _), [_, _, _ | _] ).
```

The point of these examples is that we can specify objects of interest not by their content, but by their structure. We only indicate their structure and leave their arguments as unspecified slots.

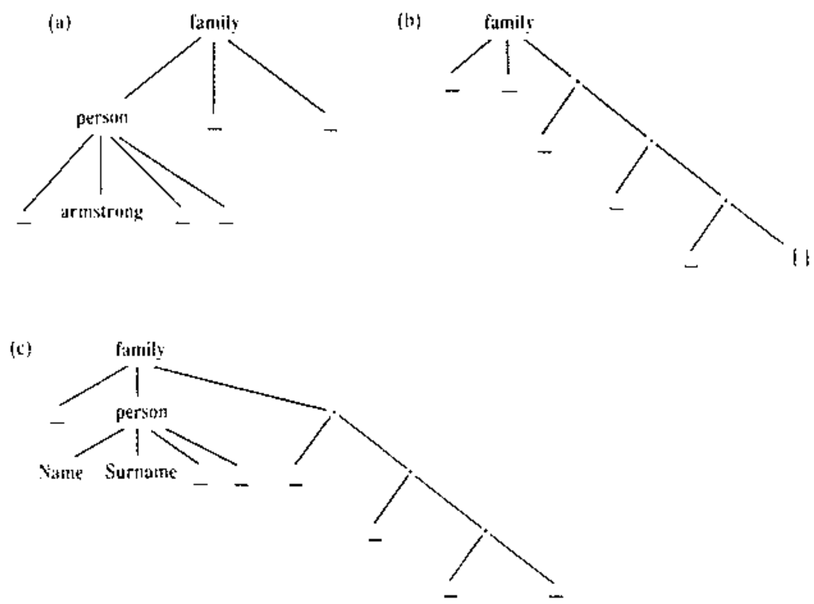


Figure 4.2 Specifying objects by their structural properties: (a) any Armstrong family; (b) any family with exactly three children; (c) any family with at least three children. Structure (c) makes provision for retrieving the wife's name through the instantiation of the variables Name and Surname.

We can provide a set of procedures that can serve as a utility to make the interaction with the database more comfortable. Such utility procedures could be part of the user interface. Some useful utility procedures for our database are:

```
husband( X) :-                    % X is a husband
  family( X, _, _).

wife( X) :-                       % X is a wife
  family( _, X, _).

child( X) :-                      % X is a child
  family( _, _, Children),
  member( X, Children).           % X in list Children

exists( Person) :-                % Any person in the database
  husband( Person)
  ;
  wife( Person)
  ;
  child( Person).
```



```
dateofbirth( person( _, _, Date, _), Date).

salary( person( _, _, _, works( _, S) ), S).     % Salary of working person

salary( person( _, _, _, unemployed), 0).        % Salary of unemployed
```

We can use these utilities, for example, in the following queries to the database:

- Find the names of all the people in the database:

```
?- exists( person( Name, Surname, _, _) ).
```

- Find all children born in 2000:

```
?- child( X),
   dateofbirth( X, date( _, _, 2000) ).
```

- Find all employed wives:

```
?- wife( person( Name, Surname, _, works( _, _) ) ).
```

- Find the names of unemployed people who were born before 1973:

```
?- exists( person( Name, Surname, date( _, _, Year), unemployed) ),
   Year < 1973.
```

- Find people born before 1960 whose salary is less than 8000:

```
?- exists( Person),
   dateofbirth( Person, date( _, _, Year) ),
   Year < 1960,
   salary( Person, Salary),
   Salary < 8000.
```

- Find the names of families with at least three children:

```
?- family( person( _, Name, _, _), _, [_, _, _ | _]).
```

To calculate the total income of a family it is useful to define the sum of salaries of a list of people as a two-argument relation:

```
total( List_of_people, Sum_of_their_salaries)
```

This relation can be programmed as:

```
total( [], 0).                           % Empty list of people

total( [Person | List], Sum) :-
  salary( Person, S),                    % S: salary of first person
  total( List, Rest),                    % Rest: sum of salaries of others
  Sum is S + Rest.
```

The total income of families can then be found by the question:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income).
```

Let the length relation count the number of elements of a list, as defined in Section 3.4. Then we can specify all families that have an income per family member of less than 2000 by:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income),
   length( [Husband, Wife | Children], N),        % N: size of family
   Income/N < 2000.
```

Exercises

4.1 Write queries to find the following from the family database:

(a) names of families without children;

(b) all employed children;

(c) names of families with employed wives and unemployed husbands;

(d) all the children whose parents differ in age by at least 15 years.

4.2 Define the relation

```
twins( Child1, Child2)
```

to find twins in the family database.

4.2 Doing data abstraction

Data abstraction can be viewed as a process of organizing various pieces of information into natural units (possibly hierarchically), thus structuring the information into some conceptually meaningful form. Each such unit of information should be easily accessible in the program. Ideally, all the details of implementing such a structure should be invisible to the user of the structure – the programmer can then just concentrate on objects and relations between them. The point of the process is to make the use of information possible without the programmer having to think about the details of how the information is actually represented.

Let us discuss one way of carrying out this principle in Prolog. Consider our family example of the previous section again. Each family is a collection of pieces of information. These pieces are all clustered into natural units such as a person or a family, so they can be treated as single objects. Assume again that the family information is structured as in Figure 4.1. In the previous section, each family was represented by a Prolog clause. Here, a family will be represented as a structured object, for example:

```
FoxFamily = family( person( tom, fox, _, _), _, _)
```


Let us now define some relations through which the user can access particular components of a family without knowing the details of Figure 4.1. Such relations can be called *selectors* as they select particular components. The name of such a selector relation will be the name of the component to be selected. The relation will have two arguments: first, the object that contains the component, and second, the component itself:

```
selector_relation( Object, Component_selected)
```

Here are some selectors for the family structure:

```
husband( family( Husband, _, _), Husband).

wife( family( _, Wife, _), Wife).

children( family( _, _, ChildList), ChildList).
```

We can also define selectors for particular children:

```
firstchild( Family, First) :-
  children( Family, [First | _] ).

secondchild( Family, Second) :-
  children( Family, [_, Second | _] ).

...
```

We can generalize this to selecting the Nth child:

```
nthchild( N, Family, Child) :-
  children( Family, ChildList),
  nth_member( N, ChildList, Child).        % Nth element of a list
```

Another interesting object is a person. Some related selectors according to Figure 4.1 are:

```
firstname( person( Name, _, _, _), Name).

surname( person( _, Surname, _, _), Surname).

born( person( _, _, Date, _), Date).
```

How can we benefit from selector relations? Having defined them, we can now forget about the particular way that structured information is represented. To create and manipulate this information, we just have to know the names of the selector relations and use these in the rest of the program. In the case of complicated representations, this is easier than always referring to the representation explicitly. In our family example in particular, the user does not have to know that the children are represented as a list. For example, assume that we want to say that Tom Fox and Jim Fox belong to the same family and that Jim is the second child of Tom. Using the selector relations above, we can define two persons, call them Person1 and Person2, and the family. The following list of goals does this:

```
firstname( Person1, tom), surname( Person1, fox),      % Person1 is Tom Fox
firstname( Person2, jim), surname( Person2, fox),      % Person2 is Jim Fox
husband( Family, Person1),
secondchild( Family, Person2)
```

As a result, the variables Person1, Person2 and Family are instantiated as:

```
Person1 = person( tom, fox, _, _)
Person2 = person( jim, fox, _, _)
Family = family( person( tom, fox, _, _), _, [ _, person( jim, fox) | _])
```

The use of selector relations also makes programs easier to modify. Imagine that we would like to improve the efficiency of a program by changing the representation of data. All we have to do is to change the definitions of the selector relations, and the rest of the program will work unchanged with the new representation.

Exercise

4.3 Complete the definition of nthchild by defining the relation

```
nth_member( N, List, X)
```

which is true if X is the Nth member of List.

4.3 Simulating a non-deterministic automaton

This exercise shows how an abstract mathematical construct can be translated into Prolog. In addition, our resulting program will turn out to be much more flexible than initially intended.

A *non-deterministic finite automaton* is an abstract machine that reads as input a string of symbols and decides whether to *accept* or to *reject* the input string. An automaton has a number of *states* and it is always in one of the states. It can change its state by moving from the current state to another state. The internal structure of the automaton can be represented by a transition graph such as that in Figure 4.3. In this example, s_1, s_2, s_3 and s_4 are the *states* of the automaton. Starting from the initial state (s_1 in our example), the automaton moves from state to state while reading the input string. Transitions depend on the current input symbol, as indicated by the arc labels in the transition graph.

A transition occurs each time an input symbol is read. Note that transitions can be non-deterministic. In Figure 4.3, if the automaton is in state s_1 and the current input symbol is *a* then it can transit into s_1 or s_2. Some arcs are labelled *null* denoting the 'null symbol'. These arcs correspond to 'silent moves' of the automaton. Such a

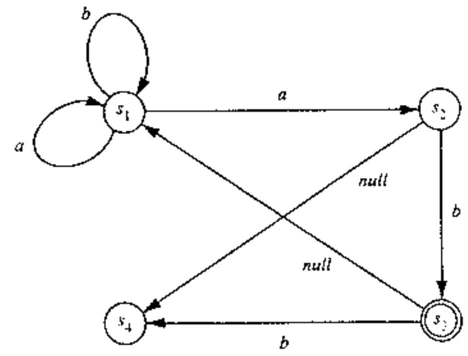


Figure 4.3 An example of a non-deterministic finite automaton.

move is said to be *silent* because it occurs without any reading of input, and the observer, viewing the automaton as a black box, will not be able to notice that any transition has occurred.

The state s_3 is double circled, which indicates that it is a *final state*. The automaton is said to *accept* the input string if there is a transition path in the graph such that

(1) it starts with the initial state,
(2) it ends with a final state, and
(3) the arc labels along the path correspond to the complete input string.

It is entirely up to the automaton to decide which of the possible moves to execute at any time. In particular, the automaton may choose to make or not to make a silent move, if it is available in the current state. But abstract non-deterministic machines of this kind have a magic property: if there is a choice then they always choose a 'right' move; that is, a move that leads to the acceptance of the input string, if such a move exists. The automaton in Figure 4.3 will, for example, accept the strings *ab* and *aabaab*, but it will reject the strings *abb* and *abba*. It is easy to see that this automaton accepts any string that terminates with *ab*, and rejects all others.

In Prolog, an automaton can be specified by three relations:

(1) a unary relation final which defines the final states of the automaton;
(2) a three-argument relation trans which defines the state transitions so that

```
trans( S1, X, S2)
```

means that a transition from a state S1 to S2 is possible when the current input symbol X is read;

(3) a binary relation

```
silent( S1, S2)
```

meaning that a silent move is possible from S1 to S2.

For the automaton in Figure 4.3 these three relations are:

```
final( s3).

trans( s1, a, s1).
trans( s1, a, s2).
trans( s1, b, s1).
trans( s2, b, s3).
trans( s3, b, s4).

silent( s2, s4).
silent( s3, s1).
```

We will represent input strings as Prolog lists. So the string *aab* will be represented by [a,a,b]. Given the description of the automaton, the simulator will process a given input string and decide whether the string is accepted or rejected. By definition, the non-deterministic automaton accepts a given string if (starting from an initial state), after having read the whole input string, the automaton can (possibly) be in its final state. The simulator is programmed as a binary relation, accepts, which defines the acceptance of a string from a given state. So

```
accepts( State, String)
```

is true if the automaton, starting from the state State as initial state, accepts the string String. The accepts relation can be defined by three clauses. They correspond to the following three cases:

(1) The empty string, [], is accepted from a state State if State is a final state.

(2) A non-empty string is accepted from State if reading the first symbol in the string can bring the automaton into some state State1, and the rest of the string is accepted from State1. Figure 4.4(a) illustrates.

(3) A string is accepted from State if the automaton can make a silent move from State to State1 and then accept the (whole) input string from State1. Figure 4.4(b) illustrates.

These rules can be translated into Prolog as:

```
accepts( State, [] ) :-                  % Accept empty string
   final( State).

accepts( State, [X | Rest] ) :-          % Accept by reading first symbol
   trans( State, X, State1),
   accepts( State1, Rest).

accepts( State, String) :-               % Accept by making silent move
   silent( State, State1),
   accepts( State1, String).
```

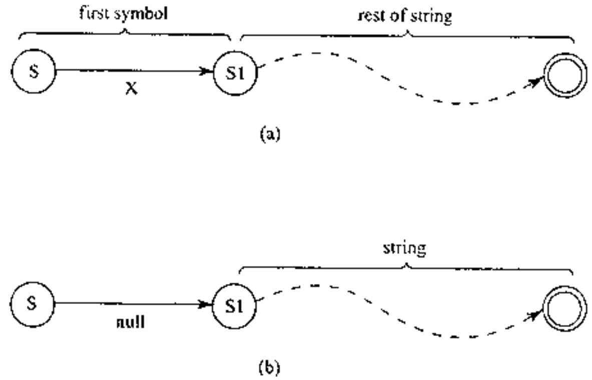



Figure 4.4 Accepting a string: (a) by reading its first symbol X; (b) by making a silent move.

The program can be asked, for example, about the acceptance of the string *aaab* by:

```
?- accepts( s1, [a,a,a,b] ).

yes
```

As we have already seen, Prolog programs are often able to solve more general problems than problems for which they were originally developed. In our case, we can also ask the simulator which state our automaton can be in initially so that it will accept the string *ab*:

```
?- accepts( S, [a,b] ).

S = s1;

S = s3
```

Amusingly, we can also ask: What are all the strings of length 3 that are accepted from state s_1?

```
?- accepts( s1, [X1,X2,X3] ).

X1 = a
X2 = a
X3 = b;

X1 = b
X2 = a
X3 = b;

no
```

If we prefer the acceptable input strings to be typed out as lists then we can formulate the question as:

```
?- String = [_, _, _], accepts( s1, String).
String = [a,a,b];
String = [b,a,b];
no
```

We can make further experiments asking even more general questions, such as: From what states will the automaton accept input strings of length 7?

Further experimentation could involve modifications in the structure of the automaton by changing the relations final, trans and silent. The automaton in Figure 4.3 does not contain any cyclic 'silent path' (a path that consists only of silent moves). If in Figure 4.3 a new transition

```
silent( s1, s3)
```

is added then a 'silent cycle' is created. But our simulator may now get into trouble. For example, the question

```
?- accepts( s1, [a] ).
```

would induce the simulator to cycle in state s_1 indefinitely, all the time hoping to find some way to the final state.

Exercises

4.4 Why could cycling not occur in the simulation of the original automaton in Figure 4.3, when there was no 'silent cycle' in the transition graph?

4.5 Cycling in the execution of accepts can be prevented, for example, by counting the number of moves made so far. The simulator would then be requested to search only for paths of some limited length. Modify the accepts relation this way. Hint: Add a third argument: the maximum number of moves allowed:

```
accepts( State, String, MaxMoves)
```

4.4 Travel agent

In this section we will construct a program that gives advice on planning air travel. The program will be a rather simple advisor, yet it will be able to answer some useful questions, such as:

- What days of the week is there a direct evening flight from Ljubljana to London?
- How can I get from Ljubljana to Edinburgh on Thursday?

- I have to visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday. In what sequence should I visit these cities so that I have no more than one flight each day of the tour?

The program will be centred around a database holding the flight information. This will be represented as a three-argument relation:

```
timetable( Place1, Place2, ListOfFlights)
```

where ListOfFlights is a list of structured items of the form:

```
DepartureTime / ArrivalTime / FlightNumber / ListOfDays
```

Here the operator '/' only holds together the components of the structure, and of course does not mean arithmetic division. ListOfDays is either a list of weekdays or the atom alldays. One clause of the timetable relation can be, for example:

```
timetable( london, edinburgh,
           [ 9:40 / 10:50 / ba4733 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
```

The times are represented as structured objects with two components, hours and minutes, combined by the operator ':'.

The main problem is to find exact routes between two given cities on a given day of the week. This will be programmed as a four-argument relation:

```
route( Place1, Place2, Day, Route)
```

Here Route is a sequence of flights that satisfies the following criteria:

(1) the start point of the route is Place1;

(2) the end point is Place2;

(3) all the flights are on the same day of the week, Day;

(4) all the flights in Route are in the timetable relation;

(5) there is enough time for transfer between flights.

The route is represented as a list of structured objects of the form:

```
From / To / FlightNumber / Departure_time
```

We will also use the following auxiliary predicates:

(1) flight(Place1, Place2, Day, FlightNum, DepTime, ArrTime)

This says that there is a flight, FlightNum, between Place1 and Place2 on the day of the week Day with the specified departure and arrival times.

(2) deptime(Route, Time)

Departure time of Route is Time.

(3) transfer(Time1, Time2)

There is at least 40 minutes between Time1 and Time2, which should be sufficient for transfer between two flights.

The problem of finding a route is reminiscent of the simulation of the non-deterministic automaton of the previous section. The similarities of both problems are as follows:

- The states of the automaton correspond to the cities.
- A transition between two states corresponds to a flight between two cities.
- The transition relation of the automaton corresponds to the timetable relation.
- The automaton simulator finds a path in the transition graph between the initial state and a final state; the travel planner finds a route between the start city and the end city of the tour.

Not surprisingly, therefore, the route relation can be defined similarly to the accepts relation, with the exception that here we have no 'silent moves'. We have two cases:

(1) Direct flight connection: if there is a direct flight between places Place1 and Place2 then the route consists of this flight only:

```
route( Place1, Place2, Day, [ Place1 / Place2 / Fnum / Dep ] ) :-
  flight( Place1, Place2, Day, Fnum, Dep, Arr).
```

(2) Indirect flight connection: the route between places P1 and P2 consists of the first flight, from P1 to some intermediate place P3, followed by a route between P3 to P2. In addition, there must be enough time between the arrival of the first flight and the departure of the second flight for transfer.

```
route( P1, P2, Day, [P1 / P3 / Fnum1 / Dep1 | RestRoute] ) :-
  route( P3, P2, Day, RestRoute),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( RestRoute, Dep2),
  transfer( Arr1, Dep2).
```

The auxiliary relations flight, transfer and deptime are easily programmed and are included in the complete travel planning program in Figure 4.5. Also included is an example timetable database.

Our route planner is extremely simple and may examine paths that obviously lead nowhere. Yet it will suffice if the flight database is not large. A really large database would require more intelligent planning to cope with the large number of potential candidate paths.

Some example questions to the program are as follows:


```
% A FLIGHT ROUTE PLANNER

:- op( 50, xfy, :).

% route( Place1, Place2, Day, Route):
%    Route is a sequence of flights on Day, starting at Place1, ending at Place2

route( P1, P2, Day, [ P1 / P2 / Fnum / Deptime ] ) :-          % Direct flight
  flight( P1, P2, Day, Fnum, Deptime, _).

route( P1, P2, Day, [ (P1/P3/Fnum1/Dep1) | RestRoute] ) :-     % Indirect connection
  route( P3, P2, Day, RestRoute),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( RestRoute, Dep2),                                   % Departure time of Route
  transfer( Arr1, Dep2).                                       % Enough time for transfer

flight( Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
  timetable( Place1, Place2, Flightlist),
  member( Deptime / Arrtime / Fnum / Daylist , Flightlist),
  flyday( Day, Daylist).

flyday( Day, Daylist) :-
  member( Day, Daylist).

flyday( Day, alldays) :-
  member( Day, [mo,tu,we,th,fr,sa,su] ).

deptime( [ P1 / P2 / Fnum / Dep | _], Dep).

transfer( Hours1:Mins1, Hours2:Mins2) :-
  60 * (Hours2 - Hours1) + Mins2 - Mins1 >= 40.

member( X, [X | L] ).

member( X, [Y | L] ) :-
  member( X, L).

% A FLIGHT DATABASE

timetable( edinburgh, london,
           [  9:40 / 10:50 / ba4733 / alldays,
             13:40 / 14:50 / ba4773 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).

timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4732 / alldays,
             11:40 / 12:50 / ba4752 / alldays,
             18:40 / 19:50 / ba4822 / [mo,tu,we,th,fr] ] ).

timetable( london, ljubljana,
           [ 13:20 / 16:20 / jp212 / [mo,tu,we,fr,su],
             16:30 / 19:30 / ba473 / [mo,we,th,sa] ] ).

timetable( london, zurich,
           [  9:10 / 11:45 / ba614 / alldays,
             14:45 / 17:20 / sr805 / alldays ] ).
```

Figure 4.5 A flight route planner and an imaginary flight timetable.

Figure 4.5 *contd*

```
timetable( london, milan,
        [  8:30 / 11:20 / ba510 / alldays,
          11:00 / 13:50 / az459 / alldays ] ).
timetable( ljubljana, zurich,
        [ 11:30 / 12:40 / jp322 / [tu,th] ] ).
timetable( ljubljana, london,
        [ 11:10 / 12:20 / jp211 / [mo,tu,we,fr,su],
          20:30 / 21:30 / ba472 / [mo,we,th,sa] ] ).
timetable( milan, london,
        [  9:10 / 10:00 / az458 / alldays,
          12:20 / 13:10 / ba511 / alldays ] ).
timetable( milan, zurich,
        [  9:25 / 10:15 / sr621 / alldays,
          12:45 / 13:35 / sr623 / alldays ] ).
timetable( zurich, ljubljana,
        [ 13:30 / 14:40 / jp323 / [tu,th] ] ).
timetable( zurich, london,
        [  9:00 /  9:40 / ba613 / [mo,tu,we,th,fr,sa],
          16:10 / 16:55 / sr806 / [mo,tu,we,th,fr,su] ] ).
timetable( zurich, milan,
        [  7:55 /  8:45 / sr620 / alldays ] ).
```

- What days of the week is there a direct evening flight from Ljubljana to London?

```
?- flight( ljubljana, london, Day, _, DeptHour: _, _), DeptHour >= 18.

Day = mo;
Day = we;
...
```

- How can I get from Ljubljana to Edinburgh on Thursday?

```
?- route( ljubljana, edinburgh, th, R).

R = [ ljubljana / zurich / jp322 / 11:30, zurich / london / sr806 / 16:10,
      london / edinburgh / ba4822 / 18:40]
```

- How can I visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday, with no more than one flight each day of the tour? This question is somewhat trickier. It can be formulated by using the permutation relation, programmed in Chapter 3. We are asking for a permutation of the cities Milan, Ljubljana and Zurich such that the corresponding flights are possible on successive days:


```
?- permutation( [milan, ljubljana, zurich], [City1, City2, City3] ),
flight( london, City1, tu, FN1, _, _),
flight( City1, City2, we, FN2, _, _),
flight( City2, City3, th, FN3, _, _),
flight( City3, london, fr, FN4, _, _).

City1 = milan
City2 = zurich
City3 = ljubljana
FN1 = ba510
FN2 = sr621
FN3 = jp323
FN4 = jp211
```

Finally let us note that this program is susceptible to indefinite loops, which happens for example if we ask it to find a route not in the timetable:

```
?- route ( moscow, edinburgh, mo, R).
```

It is better therefore to keep questions safe by limiting the length of the route. We can use the usual trick with conc:

```
?- conc( R, _, [_,_,_,_]), route( moscow, edinburgh, mo, R).

no
```

The conc goal limits the list R to length 4 and also forces the search to consider shortest routes first.

4.5 The eight queens problem

The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen. The solution will be programmed as a unary predicate

```
solution( Pos)
```

which is true if and only if Pos represents a position with eight queens that do not attack each other. It will be interesting to compare various ideas for programming this problem. Therefore we will present three programs based on somewhat different representations of the problem.

4.5.1 Program 1

First we have to choose a representation of the board position. One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen. Each item in the list will specify a square of the board on which the

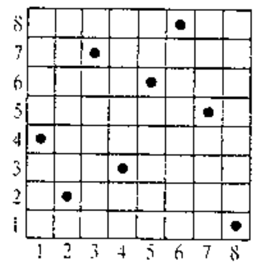


Figure 4.6 A solution to the eight queens problem. This position can be specified by the list [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1].

corresponding queen is sitting. Further, each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8. In the program we can write such a pair as:

X/Y

where, of course, the '/' operator is not meant to indicate division, but simply combines both coordinates together into a square. Figure 4.6 shows one solution of the eight queens problem and its list representation.

Having chosen this representation, the problem is to find such a list of the form:

[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

which satisfies the no-attack requirement. Our procedure solution will have to search for a proper instantiation of the variables X1, Y1, X2, Y2, ..., X8, Y8. As we know that all the queens will have to be in different columns to prevent vertical attacks, we can immediately constrain the choice and so make the search task easier. We can thus fix the X-coordinates so that the solution list will fit the following, more specific template:

[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

We are interested in the solution on a board of size 8 by 8. However, in programming, the key to the solution is often in considering a more general problem. Paradoxically, it is often the case that the solution for the more general problem is easier to formulate than that for the more specific, original problem. The original problem is then simply solved as a special case of the more general problem.

The creative part of the problem is to find the correct generalization of the original problem. In our case, a good idea is to generalize the number of queens (the number of columns in the list) from 8 to any number, including zero. The solution relation can then be formulated by considering two cases:

Case 1 The list of queens is empty: the empty list is certainly a solution because there is no attack.

Case 2 The list of queens is non-empty: then it looks like this:

[X/Y | Others]

In case 2, the first queen is at some square X/Y and the other queens are at squares specified by the list Others. If this is to be a solution then the following conditions must hold:

(1) There must be no attack between the queens in the list Others; that is, Others itself must also be a solution.

(2) X and Y must be integers between 1 and 8.

(3) A queen at square X/Y must not attack any of the queens in the list Others.

To program the first condition we can simply use the solution relation itself. The second condition can be specified as follows: Y will have to be a member of the list of integers between 1 and 8 – that is, [1,2,3,4,5,6,7,8]. On the other hand, we do not have to worry about X since the solution list will have to match the template in which the X-coordinates are already specified. So X will be guaranteed to have a proper value between 1 and 8. We can implement the third condition as another relation, noattack. All this can then be written in Prolog as follows:

```
solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).
```

It now remains to define the noattack relation:

```
noattack( Q, Qlist)
```

Again, this can be broken down into two cases:

(1) If the list Qlist is empty then the relation is certainly true because there is no queen to be attacked.

(2) If Qlist is not empty then it has the form [Q1 | Qlist1] and two conditions must be satisfied:

(a) the queen at Q must not attack the queen at Q1, and

(b) the queen at Q must not attack any of the queens in Qlist1.

To specify that a queen at some square does not attack another square is easy: the two squares must not be in the same row, the same column or the same diagonal. Our solution template guarantees that all the queens are in different columns, so it only remains to specify explicitly that:

- the Y-coordinates of the queens are different, and
- they are not in the same diagonal, either upward or downward; that is, the distance between the squares in the X-direction must not be equal to that in the Y-direction.

```
% solution( BoardPosition) if BoardPosition is a list of non-attacking queens

solution( [] ).

solution( [X/Y | Others] ) :-              % First queen at X/Y, other queens at Others
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).                  % First queen does not attack others

noattack( _, [] ).                         % Nothing to attack

noattack( X/Y, [X1/Y1 | Others] ) :-
  Y =\= Y1,                                % Different Y-coordinates
  Y1 - Y =\= X1 - X,                       % Different diagonals
  Y1 - Y =\= X - X1,
  noattack( X/Y, Others).

member( Item, [Item | Rest] ).

member( Item, [First | Rest] ) :-
  member( Item, Rest).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 4.7 Program 1 for the eight queens problem.

Figure 4.7 shows the complete program. To alleviate its use a template list has been added. This list can be retrieved in a question for generating solutions. So we can now ask:

```
?- template( S), solution( S).
```

and the program will generate solutions as follows:

```
S = [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [ 1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [ 1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Exercise

4.6 When searching for a solution, the program of Figure 4.7 explores alternative values for the Y-coordinates of the queens. At which place in the program is the order of alternatives defined? How can we easily modify the program to change the order? Experiment with different orders with the view of studying the time efficiency of the program.

4.5.2 Program 2

In the board representation of program 1, each solution had the form

```
[ 1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

because the queens were simply placed in consecutive columns. No information is lost if the X-coordinates were omitted. So a more economical representation of the board position can be used, retaining only the Y-coordinates of the queens:

```
[ Y1, Y2, Y3, ..., Y8]
```

To prevent the horizontal attacks, no two queens can be in the same row. This imposes a constraint on the Y-coordinates. The queens have to occupy all the rows 1, 2, ..., 8. The choice that remains is the *order* of these eight numbers. Each solution is therefore represented by a permutation of the list

```
[1,2,3,4,5,6,7,8]
```

Such a permutation, S, is a solution if all the queens are safe. So we can write:

```
solution( S) :-
  permutation( [1,2,3,4,5,6,7,8], S),
  safe( S).
```

We have already programmed the permutation relation in Chapter 3, but the safe relation remains to be specified. We can split its definition into two cases:

(1) S is the empty list: this is certainly safe as there is nothing to be attacked.
(2) S is a non-empty list of the form [Queen | Others]. This is safe if the list Others is safe, and Queen does not attack any queen in the list Others.

In Prolog, this is:

```
safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others).
```

The noattack relation here is slightly trickier. The difficulty is that the queens' positions are only defined by their Y-coordinates, and the X-coordinates are not explicitly present. This problem can be circumvented by a small generalization of the noattack relation, as illustrated in Figure 4.8. The goal

```
noattack( Queen,Others)
```

is meant to ensure that Queen does not attack Others when the X-distance between Queen and Others is equal to 1. What is needed is the generalization of the X-distance between Queen and Others. So we add this distance as the third argument of the noattack relation:

```
noattack( Queen, Others, Xdist)
```

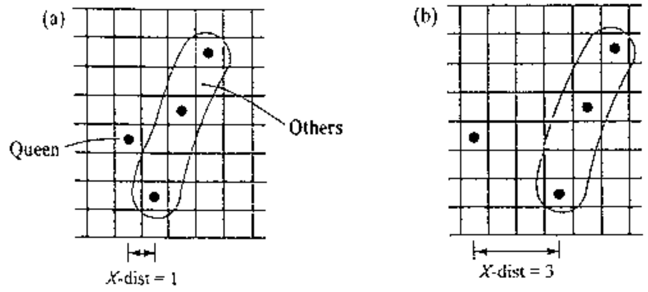


Figure 4.8 (a) X-distance between Queen and Others is 1. (b) X-distance between Queen and Others is 3.

Accordingly, the noattack goal in the safe relation has to be modified to

```
noattack( Queen, Others, 1)
```

The noattack relation can now be formulated according to two cases, depending on the list Others: if Others is empty then there is no target and certainly no attack; if Others is non-empty then Queen must not attack the first queen in Others (which is Xdist columns from Queen) and also the tail of Others at Xdist + 1. This leads to the program shown in Figure 4.9.

4.5.3 Program 3

Our third program for the eight queens problem will be based on the following reasoning. Each queen has to be placed on some square; that is, into some column, some row, some upward diagonal and some downward diagonal. To make sure that all the queens are safe, each queen must be placed in a different column, a different row, a different upward and a different downward diagonal. It is thus natural to consider a richer representation with four coordinates:

- x columns
- y rows
- u upward diagonals
- v downward diagonals

The coordinates are not independent: given x and y, u and v are determined (Figure 4.10 illustrates). For example, as:

$$u = x - y$$
$$v = x + y$$


```
% solution( Queens) if Queens is a list of Y-coordinates of eight non-attacking queens

solution( Queens) :-
  permutation( [1,2,3,4,5,6,7,8], Queens),
  safe( Queens).

permutation( [], [] ).

permutation( [Head | Tail], PermList) :-
  permutation( Tail, PermTail),
  del( Head, PermList, PermTail).                    % Insert Head in permuted Tail

% del( Item, List, NewList): deleting Item from List gives NewList

del( Item, [Item | List], List).

del( Item, [First | List], [First | List1] ) :-
  del( Item, List, List1).

% safe( Queens) if Queens is a list of Y-coordinates of non-attacking queens

safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others, 1).

noattack( _, [], _).

noattack( Y, [Y1 | Ylist], Xdist) :-
  Y1 - Y =\= Xdist,
  Y - Y1 =\= Xdist,
  Dist1 is Xdist + 1,
  noattack( Y, Ylist, Dist1).
```

Figure 4.9 Program 2 for the eight queens problem.

The domains for all four dimensions are:

Dx = [1,2,3,4,5,6,7,8]

Dy = [1,2,3,4,5,6,7,8]

Du = [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]

Dv = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

The eight queens problem can now be stated as follows: select eight 4-tuples (X,Y,U,V) from the domains (X from Dx, Y from Dy, etc.), never using the same element twice from any of the domains. Of course, once X and Y are chosen, U and V are determined. The solution can then be, roughly speaking, as follows: given all four domains, select the position of the first queen, delete the corresponding items from the four domains, and then use the rest of the domains for placing the rest of the queens. A program based on this idea is shown in Figure 4.11. The board

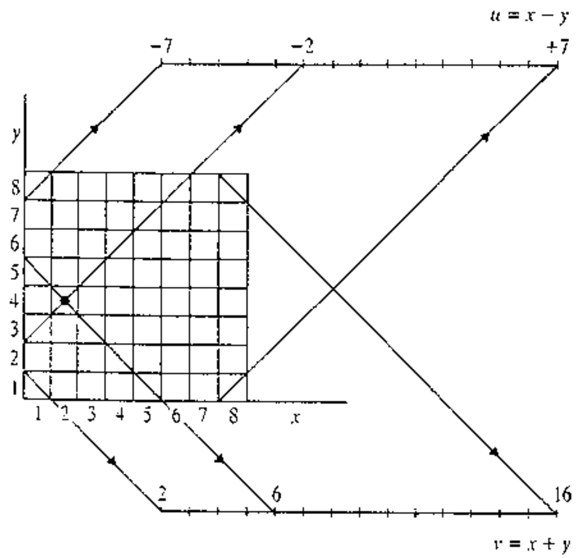


Figure 4.10 The relation between columns, rows, upward and downward diagonals. The indicated square has coordinates: $x = 2$, $y = 4$, $u = 2 - 4 = -2$, $v = 2 + 4 = 6$.

```
% solution( Ylist) if Ylist is a list of Y-coordinates of eight non-attacking queens

solution( Ylist) :-
  sol( Ylist,                                     % Y-coordinates of queens
    [1,2,3,4,5,6,7,8],                            % Domain for X-coordinates
    [1,2,3,4,5,6,7,8],                            % Domain for Y-coordinates
    [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],       % Upward diagonals
    [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ).     % Downward diagonals

sol( [], [], Dy, Du, Dv).

sol( [Y | Ylist], [X | Dx1], Dy, Du, Dv) :-
  del( Y, Dy, Dy1),                               % Choose a Y-coordinate
  U is X-Y,                                       % Corresponding upward diagonal
  del( U, Du, Du1),                               % Remove it
  V is X+Y,                                       % Corresponding downward diagonal
  del( V, Dv, Dv1),                               % Remove it
  sol( Ylist, Dx1, Dy1, Du1, Dv1).                % Use remaining values

del( Item, [Item | List], List).

del( Item, [First | List], [First | List1] ) :-
  del( Item, List, List1).
```

Figure 4.11 Program 3 for the eight queens problem.

position is, again, represented by a list of Y-coordinates. The key relation in this program is

```
sol( Ylist, Dx, Dy, Du, Dv)
```

which instantiates the Y-coordinates (in Ylist) of the queens, assuming that they are placed in consecutive columns taken from Dx. All Y-coordinates and the corresponding U and V-coordinates are taken from the lists Dy, Du and Dv. The top procedure, solution, can be invoked by the question:

```
?- solution( S).
```

This will cause the invocation of sol with the complete domains that correspond to the problem space of eight queens.

The sol procedure is general in the sense that it can be used for solving the N-queens problem (on a chessboard of size N by N). It is only necessary to properly set up the domains Dx, Dy, etc.

It is practical to mechanize the generation of the domains. For that we need a procedure

```
gen( N1, N2, List)
```

which will, for two given integers N1 and N2, produce the list:

```
List = [ N1, N1 + 1, N1 + 2, ..., N2 - 1, N2]
```

Such a procedure is:

```
gen( N, N, [N] ).

gen( N1, N2, [N1 | List] ) :-
  N1 < N2,
  M is N1 + 1,
  gen( M, N2, List).
```

The top level relation, solution, has to be accordingly generalized to

```
solution( N, S)
```

where N is the size of the board and S is a solution represented as a list of Y-coordinates of N queens. The generalized solution relation is:

```
solution( N, S) :-
  gen( 1, N, Dxy),                 % Dxy - domain for X and Y
  Nu1 is 1 - N, Nu2 is N - 1,
  gen( Nu1, Nu2, Du),
  Nv2 is N + N,
  gen( 2, Nv2, Dv),
  sol( S, Dxy, Dxy, Du, Dv).
```

For example, a solution to the 12-queens problem would be generated by:

```
?- solution( 12, S).
S = [1,3,5,8,10,12,6,11,2,7,9,4]
```

4.5.4 Concluding remarks

The three solutions to the eight queens problem show how the same problem can be approached in different ways. We also varied the representation of data. Sometimes the representation was more economical, sometimes it was more explicit and partially redundant. The drawback of the more economical representation is that some information always has to be recomputed when it is required.

At several points, the key step toward the solution was to generalize the problem. Paradoxically, by considering a more general problem, the solution became easier to formulate. This generalization principle is a kind of standard technique that can often be applied.

Of the three programs, the third one illustrates best how to approach general problems of constructing under constraints a structure from a given set of elements.

A natural question is: Which of the three programs is most efficient? In this respect, program 2 is far inferior while the other two programs are similar. The reason is that permutation-based program 2 constructs complete permutations while the other two programs are able to recognize and reject unsafe permutations when they are only partially constructed. Program 3 avoids some of the arithmetic computation that is essentially captured in the redundant board representation this program uses.

Exercise

4.7 Let the squares of the chessboard be represented by pairs of their coordinates of the form X/Y, where both X and Y are between 1 and 8.

(a) Define the relation jump(Square1, Square2) according to the knight jump on the chessboard. Assume that Square1 is always instantiated to a square while Square2 can be uninstantiated. For example:

```
?- jump( 1/1, S).

S = 3/2;

S = 2/3;

no
```

(b) Define the relation knightpath(Path) where Path is a list of squares that represent a legal path of a knight on the empty chessboard.

(c) Using this knightpath relation, write a question to find any knight's path of length 4 moves from square 2/1 to the opposite edge of the board (Y = 8) that goes through square 5/4 after the second move.

Summary

The examples of this chapter illustrate some strong points and characteristic features of Prolog programming:

- A database can be naturally represented as a set of Prolog facts.
- Prolog's mechanisms of querying and matching can be flexibly used for retrieving structured information from a database. In addition, utility procedures can be easily defined to further alleviate the interaction with a particular database.
- *Data abstraction* can be viewed as a programming technique that makes the use of complex data structures easier, and contributes to the clarity of programs. It is easy in Prolog to carry out the essential principles of data abstraction.
- Abstract mathematical constructs, such as automata, can often be readily translated into executable Prolog definitions.
- As in the case of eight queens, the same problem can be approached in different ways by varying the representation of the problem. Often, introducing redundancy into the representation saves computation. This entails trading space for time.
- Often, the key step toward a solution is to generalize the problem. Paradoxically, by considering a more general problem the solution may become easier to formulate.

chapter 5

Controlling Backtracking

We have already seen that a programmer can control the execution of a program through the ordering of clauses and goals. In this chapter we will look at another control facility, called 'cut', for preventing backtracking. The cut also extends the expressive power of Prolog and enables the definition of a kind of negation, called 'negation as failure' and associated with the 'closed world assumption'.

5.1 Preventing backtracking

Prolog will automatically backtrack if this is necessary for satisfying a goal. Automatic backtracking is a useful programming concept because it relieves the programmer of the burden of programming backtracking explicitly. On the other hand, uncontrolled backtracking may cause inefficiency in a program. Therefore we sometimes want to control, or to prevent, backtracking. We can do this in Prolog by using the 'cut' facility.

Let us first study the behaviour of a simple example program whose execution involves some unnecessary backtracking. We will identify those points at which the backtracking is useless and leads to inefficiency.

Consider the double-step function shown in Figure 5.1. The relation between X and Y can be specified by three rules:

Rule 1: if $X < 3$ then $Y = 0$

Rule 2: if $3 \leq X$ and $X < 6$ then $Y = 2$

Rule 3: if $6 \leq X$ then $Y = 4$

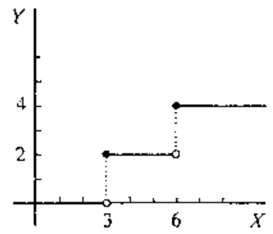


Figure 5.1 A double-step function.

This can be written in Prolog as a binary relation:

```
f( X, Y)
```

as follows:

```
f( X, 0) :- X < 3.              % Rule 1
f( X, 2) :- 3 =< X, X < 6.      % Rule 2
f( X, 4) :- 6 =< X.             % Rule 3
```

This program, of course, assumes that before f(X, Y) is executed X is already instantiated to a number, as this is required by the comparison operators.

We will make two experiments with this program. Each experiment will reveal some source of inefficiency in the program, and we will remove each source in turn by using the cut mechanism.

5.1.1 Experiment 1

Let us analyze what happens when the following question is posed:

```
?- f( 1, Y), 2 < Y.
```

When executing the first goal, f(1, Y), Y becomes instantiated to 0. So the second goal becomes

```
2 < 0
```

which fails, and so does the whole goal list. This is straightforward, but before admitting that the goal list is not satisfiable, Prolog tries, through backtracking, two useless alternatives. The detailed trace is shown in Figure 5.2.

The three rules about the f relation are mutually exclusive so that one of them at most will succeed. Therefore we, not Prolog, know that as soon as one rule succeeds there is no point in trying to use the others, as they are bound to fail. In the example of Figure 5.2, rule 1 has become known to succeed at the point indicated by 'CUT'. In order to prevent futile backtracking at this point we have to tell Prolog explicitly

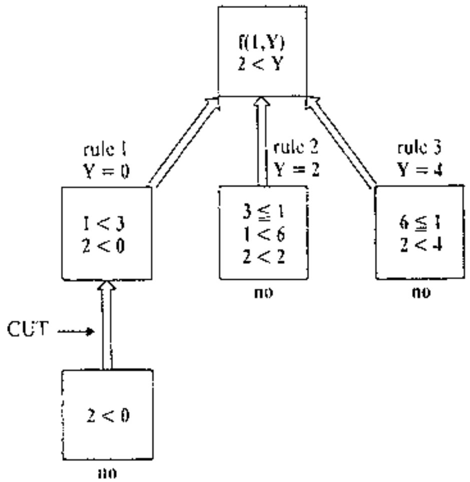


Figure 5.2 At the point marked 'CUT' we already know that the rules 2 and 3 are bound to fail.

not to backtrack. We can do this by using the cut mechanism. The 'cut' is written as ! and is inserted between goals as a kind of pseudo-goal. Our program, rewritten with cuts, is:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3 =< X, X < 6, !.
f( X, 4) :- 6 =< X.
```

The ! symbol will now prevent backtracking at the points at which it appears in the program. If we now ask:

```
?- f( 1, Y), 2 < Y.
```

Prolog will produce the same left-hand branch as in Figure 5.2. This branch will fail at the goal 2 < 0. Now Prolog will try to backtrack, but not beyond the point marked ! in the program. The alternative branches that correspond to 'rule 2' and 'rule 3' will not be generated.

The new program, equipped with cuts, is in general more efficient than the original version without cuts. When the execution fails, the new program will in general recognize this sooner than the original program.

To conclude, we have improved the efficiency by adding cuts. If the cuts are now removed in this example, the program will still produce the same result; it will perhaps only spend more time. In our case, by introducing the cut we only changed the procedural meaning of the program; that is, the results of the program were not affected. We will see later that using a cut may affect the results as well.

5.1.2 Experiment 2

Let us now perform a second experiment with the second version of our program. Suppose we ask:

```
?- f( 7, Y).
Y = 4
```

Let us analyze what has happened. All three rules were tried before the answer was obtained. This produced the following sequence of goals:

Try rule 1: 7 < 3 fails, backtrack and try rule 2 (cut was not reached)

Try rule 2: 3 ≤ 7 succeeds, but then 7 < 6 fails, backtrack and try rule 3 (cut was not reached)

Try rule 3: 6 ≤ 7 succeeds

This trace reveals another source of inefficiency. First it is established that X < 3 is not true (7 < 3 fails). The next goal is 3 =< X (3 =< 7 succeeds). But we know that once the first test has failed the second test is bound to succeed as it is the negation of the first. Therefore the second test is redundant and the corresponding goal can be omitted. The same is true about the goal 6 =< X in rule 3. This leads to the following, more economical formulation of the three rules:

if X < 3 then Y = 0,
otherwise if X < 6 then Y = 2,
otherwise Y = 4.

We can now omit the conditions in the program that are guaranteed to be true whenever they are executed. This leads to the third version of the program:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- X < 6, !.
f( X, 4).
```

This program produces the same results as our original version, but is more efficient than both previous versions. But what happens if we *now* remove the cuts? The program becomes:

```
f( X, 0) :- X < 3.
f( X, 2) :- X < 6.
f( X, 4).
```

This may produce multiple solutions, some of which are not correct. For example:

```
?- f( 1, Y).
Y = 0;
```

```
Y = 2;
Y = 4;
no
```

It is important to notice that, in contrast to the second version of the program, this time the cuts do not only affect the procedural behaviour, but also change the results of the program.

A more precise meaning of the cut mechanism is as follows:

Let us call the 'parent goal' the goal that matched the head of the clause containing the cut. When the cut is encountered as a goal it succeeds immediately, but it commits the system to all choices made between the time the 'parent goal' was invoked and the time the cut was encountered. All the remaining alternatives between the parent goal and the cut are discarded.

To clarify this definition consider a clause of the form:

```
H :- B1, B2, ..., Bm, !, ..., Bn.
```

Let us assume that this clause was invoked by a goal G that matched H. Then G is the parent goal. At the moment that the cut is encountered, the system has already found some solution of the goals B1, ..., Bm. When the cut is executed, this (current) solution of B1, ..., Bm becomes frozen and all possible remaining alternatives are discarded. Also, the goal G now becomes committed to this clause: any attempt to match G with the head of some other clause is precluded.

Let us apply these rules to the following example:

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
?- A.
```

Here A, B, C, D, P, etc. have the syntax of terms. The cut will affect the execution of the goal C as illustrated by Figure 5.3. Backtracking will be possible within the goal list P, Q, R; however, as soon as the cut is reached, all alternative solutions of the goal list P, Q, R are suppressed. The alternative clause about C,

```
C :- V.
```

will also be discarded. However, backtracking will still be possible within the goal list S, T, U. The 'parent goal' of the clause containing the cut is the goal C in the clause:

```
A :- B, C, D.
```

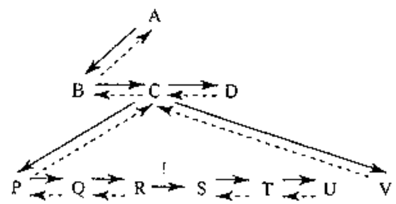



Figure 5.3 The effect of the cut on the execution. Starting with A, the solid arrows indicate the sequence of calls; the dashed arrows indicate backtracking. There is 'one way traffic' between R and S.

Therefore the cut will only affect the execution of the goal C. On the other hand, it will be 'invisible' from goal A. So automatic backtracking within the goal list B, C, D will remain active regardless of the cut within the clause used for satisfying C.

5.2 Examples using cut

5.2.1 Computing maximum

The procedure for finding the larger of two numbers can be programmed as a relation

```
max( X, Y, Max)
```

where Max = X if X is greater than or equal to Y, and Max is Y if X is less than Y. This corresponds to the following two clauses:

```
max( X, Y, X) :- X >= Y.
max( X, Y, Y) :- X < Y.
```

These two rules are mutually exclusive. If the first one succeeds then the second one will fail. If the first one fails then the second must succeed. Therefore a more economical formulation, with 'otherwise', is possible:

If $X \geqq Y$ then Max = X,
otherwise Max = Y.

This is written in Prolog using a cut as:

```
max( X, Y, X) :- X >= Y, !.
max( X, Y, Y).
```

It should be noted that the use of this procedure requires care. It is safe if in the goal max(X,Y,Max) the argument Max is not instantiated. The following example of incorrect use illustrates the problem:

```
?- max( 3, 1, 1).

yes
```

The following reformulation of max overcomes this limitation:

```
max( X, Y, Max) :-
  X >= Y, !, Max = X
  ;
  Max = Y.
```

5.2.2 Single-solution membership

We have been using the relation

```
member( X, L)
```

for establishing whether X is in list L. The program was:

```
member( X, [X | L] ).

member( X, [Y | L] ) :- member( X, L).
```

This is non-deterministic: if X occurs several times then any occurrence can be found. Let us now change member into a deterministic procedure which will find only the first occurrence. The change is simple: we only have to prevent backtracking as soon as X is found, which happens when the first clause succeeds. The modified program is:

```
member( X, [X | L] ) :- !.

member( X, [Y | L] ) :- member( X, L).
```

This program will generate just one solution. For example:

```
?- member( X, [a,b,c] ).

X = a;

no
```

5.2.3 Adding an element to a list without duplication

Often we want to add an item X to a list L so that X is added only if X is not yet in L. If X is already in L then L remains the same because we do not want to have redundant duplicates in L. The add relation has three arguments:

```
add( X, L, L1)
```


where X is the item to be added, L is the list to which X is to be added and L1 is the resulting new list. Our rule for adding can be formulated as:

If X is a member of list L then L1 = L,
otherwise L1 is equal to L with X inserted.

It is easiest to insert X in front of L so that X becomes the head of L1. This is then programmed as follows:

```
add( X, L, L) :- member( X, L), !.

add( X, L, [X | L] ).
```

The behaviour of this procedure is illustrated by the following example:

```
?- add( a, [b,c], L).

L = [a,b,c]

?- add( X, [b,c], L).

L = [b,c]
X = b

?- add( a, [b,c,X], L).

L = [b,c,a]
X = a
```

Similar to the foregoing example with max, add(X, L1, L2) is intended to be called with L2 uninstantiated. Otherwise the result may be unexpected: for example add(a, [a], [a,a]) succeeds.

This example is instructive because we cannot easily program the 'non-duplicate add' without the use of cut or another construct derived from the cut. If we omit the cut in the foregoing program then the add relation will also add duplicate items. For example:

```
?- add( a, [a,b,c], L).

L = [a,b,c];

L = [a,a,b,c]
```

So the cut is necessary here to specify the intended relation, and not only to improve efficiency. The next example also illustrates this point.

5.2.4 Classification into categories

Assume we have a database of results of tennis games played by members of a club. The pairings were not arranged in any systematic way, so each player just played some other players. The results are in the program represented as facts like:

```
beat( tom, jim).
beat( ann, tom).
beat( pat, jim).
```

We want to define a relation

```
class( Player, Category)
```

that ranks the players into categories. We have just three categories:

winner: every player who won all his or her games is a winner
fighter: any player that won some games and lost some
sportsman: any player who lost all his or her games

For example, if all the results available are just those above then Ann and Pat are winners, Tom is a fighter and Jim is a sportsman.

It is easy to specify the rule for a fighter:

X is a fighter if
  there is some Y such that X beat Y and
  there is some Z such that Z beat X.

Now a rule for a winner:

X is a winner if
  X beat some Y and
  X was not beaten by anybody.

This formulation contains 'not' which cannot be directly expressed with our present Prolog facilities. So the formulation of winner appears trickier. The same problem occurs with sportsman. The problem can be circumvented by combining the definition of winner with that of fighter, and using the 'otherwise' connective. Such a formulation is:

If X beat somebody and X was beaten by somebody
  then X is a fighter,
  otherwise if X beat somebody
    then X is a winner,
    otherwise if X got beaten by somebody
      then X is a sportsman.

This formulation can be readily translated into Prolog. The mutual exclusion of the three alternative categories is indicated by the cuts:

```
class( X, fighter) :-
  beat( X, _),
  beat( _, X), !.

class( X, winner) :-
  beat( X, _), !.
```



```
class( X, sportsman) :-
  beat( _, X).
```

Notice that the cut in the clause for winner is not necessary. Care is needed when using such procedures containing cuts. Here is what can happen:

```
?- class( tom, C).
C = fighter;                % As intended
no
?- class( tom, sportsman).
yes                         % Not as intended
```

The call of class is safe if the second argument is not instantiated. Otherwise we may get an unintended result.

Exercises

5.1 Let a program be:

```
p( 1).
p( 2) :- !.
p( 3).
```

Write all Prolog's answers to the following questions:

(a) ?- p(X).

(b) ?- p(X), p(Y).

(c) ?- p(X), !, p(Y).

5.2 The following relation classifies numbers into three classes: positive, zero and negative:

```
class( Number, positive) :- Number > 0.
class( 0, zero).
class( Number, negative) :- Number < 0.
```

Define this procedure in a more efficient way using cuts.

5.3 Define the procedure

```
split( Numbers, Positives, Negatives)
```

which splits a list of numbers into two lists: positive ones (including zero) and negative ones. For example:

```
split( [3,-1,0,5,-2], [3,0,5], [-1,-2] )
```

Propose two versions: one with a cut and one without.

5.3 Negation as failure

'Mary likes all animals but snakes'. How can we say this in Prolog? It is easy to express one part of this statement: Mary likes any X if X is an animal. This is in Prolog:

```
likes( mary, X) :- animal( X).
```

But we have to exclude snakes. This can be done by using a different formulation:

If X is a snake then 'Mary likes X' is not true,
otherwise if X is an animal then Mary likes X.

That something is not true can be said in Prolog by using a special goal, fail, which always fails, thus forcing the parent goal to fail. The above formulation is translated into Prolog, using fail, as follows:

```
likes( mary, X) :-
  snake( X), !, fail.

likes( mary, X) :-
  animal( X).
```

The first rule here will take care of snakes: if X is a snake then the cut will prevent backtracking (thus excluding the second rule) and fail will cause the failure. These two clauses can be written more compactly as one clause:

```
likes( mary, X) :-
  snake( X), !, fail
  ;
  animal( X).
```

We can use the same idea to define the relation

```
different( X, Y)
```

which is true if X and Y are different. We have to be more precise, however, because 'different' can be understood in several ways:

- X and Y are not literally the same;
- X and Y do not match;
- the values of arithmetic expressions X and Y are not equal.

Let us choose here that X and Y are different if they do not match. The key to saying this in Prolog is:

If X and Y match then different(X, Y) fails,
otherwise different(X, Y) succeeds.

We again use the cut and fail combination:

```
different( X, X) :- !, fail.

different( X, Y).
```

This can also be written as one clause:

```
different( X, Y) :-
  X = Y, !, fail
  ;
  true.
```

true is a goal that always succeeds.

These examples indicate that it would be useful to have a unary predicate 'not' such that

```
not( Goal)
```

is true if Goal is not true. We will now define the not relation as follows:

If Goal succeeds then not(Goal) fails,
otherwise not(Goal) succeeds.

This definition can be written in Prolog as:

```
not( P) :-
  P, !, fail
  ;
  true.
```

Henceforth, we will assume that not is a built-in Prolog procedure that behaves as defined here. We will also assume that not is defined as a prefix operator, so that we can also write the goal

```
not( snake(X) )
```

as:

```
not snake( X)
```

Some Prolog implementations, in fact, support this notation. If not, then we can always define not ourselves. Alternatively, not Goal is written as \+ Goal. This more mysterious notation is also recommended in the Prolog standard for the following reason. not defined as failure, as here, does not exactly correspond to negation in mathematical logic. This difference can cause unexpected behaviour if not is used without care. This will be discussed later in the chapter.

Nevertheless, not is a useful facility and can often be used advantageously in place of cut. Our two examples can be rewritten with not as:

```
likes( mary, X) :-
  animal( X),
  not snake( X).
```

```
solution( [] ).

solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),            % Usual member predicate
  not attacks( X/Y, Others).

attacks( X/Y, Others) :-
  member( X1/Y1, Others),
  ( Y1 = Y;
    Y1 is Y + X1 - X;
    Y1 is Y - X1 + X ).
```

Figure 5.4 Another eight queens program.

```
different( X, Y) :-
  not( X = Y).
```

This certainly looks better than our original formulations. It is more natural and is easier to read.

Our tennis classification program of the previous section can also be rewritten, using not, in a way that is closer to the initial definition of the three categories:

```
class( X, fighter) :-
  beat( X, _),
  beat( _, X).

class( X, winner) :-
  beat( X, _),
  not beat( _, X).

class( X, sportsman) :-
  beat( _, X),
  not beat( X, _).
```

As another example of the use of not let us reconsider program 1 for the eight queens problem of the previous chapter (Figure 4.7). We specified the no_attack relation between a queen and other queens. This relation can be formulated also as the negation of the attack relation. Figure 5.4 shows a program modified accordingly.

Exercises

5.4 Given two lists, Candidates and RuledOut, write a sequence of goals (using member and not) that will through backtracking find all the items in Candidates that are not in RuledOut.

5.5 Define the set subtraction relation

```
set_difference( Set1, Set2, SetDifference)
```

where all the three sets are represented as lists. For example:

```
set_difference( [a,b,c,d], [b,d,e,f], [a,c] )
```

5.6 Define the predicate

```
unifiable( List1, Term, List2)
```

where List2 is the list of all the members of List1 that match Term, but are not instantiated by this matching. For example:

```
?- unifiable( [X, b, t(Y)], t(a), List).
List = [X, t(Y)]
```

Note that X and Y have to remain uninstantiated although the matching with t(a) does cause their instantiation. Hint: Use not(Term1 = Term2). If Term1 = Term2 succeeds then not(Term1 = Term2) fails and the resulting instantiation is undone!

5.4 Problems with cut and negation

Using the cut facility we get something, but not for nothing. The advantages and disadvantages of using cut were illustrated by examples in the previous sections. Let us summarize, first the advantages:

(1) With cut we can often improve the efficiency of the program. The idea is to explicitly tell Prolog: do not try other alternatives because they are bound to fail.

(2) Using cut we can specify mutually exclusive rules; so we can express rules of the form:

if condition P *then* conclusion Q,
otherwise conclusion R

In this way, cut enhances the expressive power of the language.

The reservations against the use of cut stem from the fact that we can lose the valuable correspondence between the declarative and procedural meaning of programs. If there is no cut in the program we can change the order of clauses and goals, and this will only affect the efficiency or termination of the program, not the declarative meaning. On the other hand, in programs with cuts, a change in the order of clauses may affect the declarative meaning. This means that we can get different results. The following example illustrates:

```
p :- a, b.
p :- c.
```

The declarative meaning of this program is: p is true if and only if a and b are both true or c is true. This can be written as a logic formula:

$$p \iff (a \,\&\, b) \vee c$$

We can change the order of the two clauses and the declarative meaning remains the same. Let us now insert a cut:

```
p :- a, !, b.
p :- c.
```

The declarative meaning is now:

$$p \iff (a \,\&\, b) \vee (\sim a \,\&\, c)$$

If we swap the clauses,

```
p :- c.
p :- a, !, b.
```

then the meaning becomes:

$$p \iff c \vee (a \,\&\, b)$$

The important point is that when we use the cut facility we have to pay more attention to the procedural aspects. Unfortunately, this additional difficulty increases the probability of a programming error.

In our examples in the previous sections we have seen that sometimes the removal of a cut from the program can change the declarative meaning of the program. But there were also cases in which the cut had no effect on the declarative meaning. The use of cuts of the latter type is less delicate, and therefore cuts of this kind are sometimes called 'green cuts'. From the point of view of readability of programs, green cuts are 'innocent' and their use is quite acceptable. When reading a program, green cuts can simply be ignored.

On the contrary, cuts that do affect the declarative meaning are called 'red cuts'. Red cuts are the ones that make programs hard to understand, and they should be used with special care.

Cut is often used in combination with a special goal, fail. In particular, we defined the negation of a goal (not) as the failure of the goal. The negation, so defined, is just a special, more restricted way of using cut. For reasons of clarity we will prefer to use not instead of the *cut–fail* combination (whenever possible), because the negation is intuitively clearer than the *cut–fail* combination.

It should be noted that not may also cause problems, and so should also be used with care. The problem is that not, as defined here, does not correspond exactly to negation in mathematics. If we ask Prolog:


```
?- not human( mary).
```

Prolog will probably answer 'yes'. But this should not be understood as Prolog saying 'Mary is not human'. What Prolog really means to say is: 'There is not enough information in the program to prove that Mary is human'. This arises because when processing a not goal, Prolog does not try to prove this goal directly. Instead, it tries to prove the opposite, and if the opposite cannot be proved then Prolog assumes that the not goal succeeds.

Such reasoning is based on the so-called *closed world assumption*. According to this assumption *the world is closed* in the sense that everything that exists is stated in the program or can be derived from the program. Accordingly then, if something is not in the program (or cannot be derived from it) then it is not true and consequently its negation is true. This deserves special care because we do not normally assume that 'the world is closed'. When we do not explicitly enter the clause

```
human( mary).
```

into our program, we do not mean to imply that Mary is not human.

To further study the special care that not requires, consider the following example about restaurants:

```
good_standard( jeanluis).
expensive( jeanluis).
good_standard( francesco).
reasonable( Restaurant) :-
  not expensive( Restaurant).
```

If we ask:

```
?- good_standard( X), reasonable( X).
```

Prolog will answer:

```
X = francesco
```

If we ask apparently the same question

```
?- reasonable( X), good_standard( X).
```

then Prolog will answer:

```
no
```

The reader is invited to trace the program to understand why we get different answers. The key difference between both questions is that the variable X is, in the first case, already instantiated when reasonable(X) is executed, whereas X is not yet instantiated in the second case. The general hint is: not Goal works safely if the variables in Goal are instantiated at the time not Goal is called. Otherwise we may get unexpected results due to reasons explained in the sequel.

The problem with uninstantiated negated goals arises from unfortunate change of the quantification of variables in negation as failure. In the usual interpretation in Prolog, the question:

```
?- expensive( X).
```

means: Does there *exist* X such that expensive(X) is true? If yes, what is X? So X is *existentially* quantified. Accordingly Prolog answers X = jeanluis. But the question:

```
?- not expensive( X).
```

is not interpreted as: Does there exist X such that not expensive(X)? The expected answer would be X = francesco. But Prolog answers 'no' because negation as failure changes the quantification to universal. The question not expensive(X) is interpreted as:

not(exists X such that expensive(X))

This is equivalent to:

For *all* X: not expensive(X)

We have discussed problems with cut, which also indirectly occur in not, in detail. The intention has been to warn users about the necessary care, not to definitely discourage the use of cut. Cut is useful and often necessary. And after all, the kind of complications that are incurred by cut in Prolog commonly occur when programming in other languages as well.

Summary

- The cut facility prevents backtracking. It is used both to improve the efficiency of programs and to enhance the expressive power of the language.
- Efficiency is improved by explicitly telling Prolog (with cut) not to explore alternatives that we know are bound to fail.
- Cut makes it possible to formulate mutually exclusive conclusions through rules of the form:

  if Condition *then* Conclusion1 *otherwise* Conclusion2
- Cut makes it possible to introduce *negation as failure*: not Goal is defined through the failure of Goal.
- Two special goals are sometimes useful: true always succeeds, fail always fails.
- There are also some reservations against cut: inserting a cut may destroy the correspondence between the declarative and procedural meaning of a program. Therefore, it is part of good programming style to use cut with care and not to use it without reason.

- not defined through failure does not exactly correspond to negation in mathematical logic. Therefore, the use of not also requires special care.

References

The distinction between 'green cuts' and 'red cuts' was proposed by van Emden (1982). Le (1993) proposes a different negation for Prolog which is mathematically advantageous, but computationally more expensive.

Le, T.V. (1993) *Techniques of Prolog Programming*. John Wiley & Sons.
van Emden, M. (1982) Red and green cuts. *Logic Programming Newsletter*: 2.

chapter 6

Input and Output

In this chapter we will investigate some built-in facilities for reading data from computer files and for outputting data to files. These procedures can also be used for formatting data objects in the program to achieve a desired output representation of these objects. We will also look at facilities for reading programs and for constructing and decomposing atoms.

6.1 Communication with files

The method of communication between the user and the program that we have been using up to now consists of user questions to the program and program answers in terms of instantiations of variables. This method of communication is simple and suffices to get the information in and out. However, it is often not quite sufficient because it is too rigid. Extensions to this basic communication method are needed in the following areas:

- input of data in forms other than questions – for example, in the form of English sentences,
- output of information in any format desired, and
- input from and output to any computer file or device and not just the user terminal.

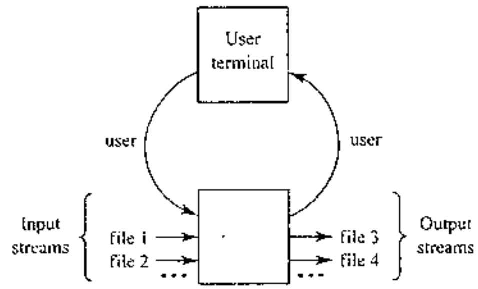


Figure 6.1 Communication between a Prolog program and several files.

Built-in predicates aimed at these extensions depend on the implementation of Prolog. We will study here a simple and handy repertoire of such predicates, which is part of many Prolog implementations. However, the implementation manual should be consulted for details and specificities. Many Prolog implementations provide various additional facilities not covered here. Such extra facilities handle windows, provide graphics primitives for drawing on the screen, input information from the mouse, and so on.

We will first consider the question of directing input and output to files, and then how data can be input and output in different forms.

Figure 6.1 shows a general situation in which a Prolog program communicates with several files. The program can, in principle, read data from several input files, also called *input streams*, and output data to several output files, also called *output streams*. Data coming from the user's terminal is treated as just another input stream. Data output to the terminal is, analogously, treated as another output stream. Both of these 'pseudo-files' are referred to by the name user. The names of other files can be chosen by the programmer according to the rules for naming files in the computer system used.

At any time during the execution of a Prolog program, only two files are 'active': one for input and one for output. These two files are called the *current input stream* and the *current output stream* respectively. At the beginning of execution these two streams correspond to the user's terminal. The current input stream can be changed to another file, Filename, by the goal:

```
see( Filename)
```

Such a goal succeeds (unless there is something wrong with Filename) and causes, as a side effect, that input is switched from the previous input stream to Filename. So a typical example of using the see predicate is the following sequence of goals, which reads something from file1 and then switches back to the terminal:

```
...
see( file1),
read_from_file( Information),
see( user),
...
```

The current output stream can be changed by a goal of the form:

```
tell( Filename)
```

A sequence of goals to output some information to file3, and then redirect succeeding output back to the terminal, is:

```
...
tell( file3),
write_on_file( Information),
tell( user),
...
```

The goal

```
seen
```

closes the current input file. The goal

```
told
```

closes the current output file.

We will assume here that files can only be processed sequentially although many Prolog implementations also handle files with random access. Sequential files behave in the same way as the terminal. Each request to read something from an input file will cause reading at the current position in the current input stream. After the reading, the current position will be, of course, moved to the next unread item. So the next request for reading will start reading at this new current position. If a request for reading is made at the end of a file, then the information returned by such a request is the atom end_of_file.

Writing is similar; each request to output information will append this information at the end of the current output stream. It is not possible to move backward and to overwrite part of the file.

We will here only consider 'text-files' – that is, files of characters. Characters are letters, digits and special characters. Some of them are said to be non-printable because when they are output on the terminal they do not appear on the screen. They may, however, have other effects, such as spacing between columns and lines.

There are two main ways in which files can be viewed in Prolog, depending on the form of information. One way is to consider the character as the basic element of the file. Accordingly, one input or output request will cause a single character to be read or written. We assume the built-in predicates for this are get, get0 and put.

The other way of viewing a file is to consider bigger units of information as basic building blocks of the file. Such a natural bigger unit is the Prolog term. So each input/output request of this type would transfer a whole term from the current input stream or to the current output stream respectively. Predicates for transfer of terms are read and write. Of course, in this case, the information in the file has to be in a form that is consistent with the syntax of terms.

What kind of file organization is chosen will, of course, depend on the problem. Whenever the problem specification will allow the information to be naturally squeezed into the syntax of terms, we will prefer to use a file of terms. It will then be possible to transfer a whole meaningful piece of information with a single request. On the other hand, there are problems whose nature dictates some other organization of files. An example is the processing of natural language sentences, say, to generate a dialogue in English between the system and the user. In such cases, files will have to be viewed as sequences of characters that cannot be parsed into terms.

6.2 Processing files of terms

6.2.1 *read* and *write*

The built-in predicate read is used for reading terms from the current input stream. The goal

```
read( X)
```

will cause the next term, T, to be read, and this term will be matched with X. If X is a variable then, as a result, X will become instantiated to T. If matching does not succeed then the goal read(X) fails. The predicate read is deterministic, so in the case of failure there will be no backtracking to input another term. Each term in the input file must be followed by a full stop and a space or carriage-return.

If read(X) is executed when the end of the current input file has been reached then X will become instantiated to the atom end_of_file.

The built-in predicate write outputs a term. So the goal

```
write( X)
```

will output the term X on the current output file. X will be output in the same standard syntactic form in which Prolog normally displays values of variables. A useful feature of Prolog is that the write procedure 'knows' to display any term no matter how complicated it may be.

Typically, there are additional built-in predicates for formatting the output. They insert spaces and new lines into the output stream. The goal

```
tab( N)
```

causes N spaces to be output. The predicate nl (which has no arguments) causes the start of a new line at output.

The following examples will illustrate the use of these procedures.

Let us assume that we have a procedure that computes the cube of a number:

```
cube( N, C) :-
  C is N * N * N.
```

Suppose we want to use this for calculating the cubes of a sequence of numbers. We could do this by a sequence of questions:

```
?- cube( 2, X).

X = 8

?- cube( 5, Y).

Y = 125

?- cube( 12, Z).

Z = 1728
```

For each number, we had to type in the corresponding goal. Let us now modify this program so that the cube procedure will read the data itself. Now the program will keep reading data and outputting their cubes until the atom stop is read:

```
cube :-
  read( X),
  process( X).

process( stop) :- !.

process( N) :-
  C is N * N * N,
  write( C),
  cube.
```

This is an example of a program whose declarative meaning is awkward to formulate. However, its procedural meaning is straightforward: to execute cube, first read X and then process it; if X = stop then everything has been done, otherwise write the cube of X and recursively call the cube procedure to process further data. A table of the cubes of numbers can be produced using this new procedure as follows:

```
?- cube.
2.
8
5.
125
12.
1728
stop.
yes
```


The numbers 2, 5 and 12 were typed in by the user on the terminal; the other numbers were output by the program. Note that each number entered by the user had to be followed by a full stop, which signals the end of a term.

It may appear that the above cube procedure could be simplified. However, the following attempt to simplify is not correct:

```
cube :-
  read( stop), !.

cube :-
  read( N),
  C is N * N * N,
  write( C),
  cube.
```

The reason why this is wrong can be seen easily if we trace the program with input data 5, say. The goal read(stop) will fail when the number is read, and this number will be lost for ever. The next read goal will input the next term. On the other hand, it could happen that the stop signal is read by the goal read(N), which would then cause a request to multiply non-numeric data.

The cube procedure conducts interaction between the user and the program. In such cases it is usually desirable that the program, before reading new data from the terminal, signals to the user that it is ready to accept the information, and perhaps also says what kind of information it is expecting. This is usually done by sending a 'prompt' signal to the user before reading. Our cube procedure would be accordingly modified, for example, as follows:

```
cube :-
  write( 'Next item, please: '),
  read( X),
  process( X).

process( stop) :- !.

process( N) :-
  C is N * N * N,
  write( 'Cube of '), write( N), write( ' is '),
  write( C), nl,
  cube.
```

A conversation with this new version of cube would then be, for example, as follows:

```
?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 12.
Cube of 12 is 1728
Next item, please: stop.
yes
```

Depending on the implementation, an extra request (like ttyflush, say) after writing the prompt might be necessary in order to force the prompt to actually appear on the screen before reading.

In the following sections we will look at some typical examples of operations that involve reading and writing.

6.2.2 Displaying lists

Besides the standard Prolog format for lists, there are several other natural forms for displaying lists which have advantages in some situations. The following procedure

```
writelist( L)
```

outputs a list L so that each element of L is written on a separate line:

```
writelist( [ ] ).

writelist( [X | L] ) :-
  write( X), nl,
  writelist( L).
```

If we have a list of lists, one natural output form is to write the elements of each list in one line. To this end, we will define the procedure writelist2. An example of its use is:

```
?- writelist2( [ [a,b,c], [d,e,f], [g,h,i] ] ).

a b c
d e f
g h i
```

A procedure that accomplishes this is:

```
writelist2( [ ] ).

writelist2( [L | LL] ) :-
  doline( L), nl,
  writelist2( LL).

doline( [ ] ).

doline( [X | L] ) :-
  write( X), tab( 1),
  doline( L).
```

A list of integer numbers can be sometimes conveniently shown as a bar graph. The following procedure, bars, will display a list in this form, assuming that the numbers in the list are sufficiently small. An example of using bars is:

```
?- bars( [3,4,6,5] ).
***
****
******
*****
```


The bars procedure can be defined as follows:

```
bars( [ ] ).

bars( [N | L] ) :-
  stars( N), nl,
  bars( L).

stars( N) :-
  N > 0,
  write( *),
  N1 is N - 1,
  stars( N1).

stars( N) :-
  N =< 0.
```

6.2.3 Processing a file of terms

A typical sequence of goals to process a whole file, F, would look something like this:

```
..., see( F), processfile, see( user), ...
```

Here processfile is a procedure to read and process each term in F, one after another, until the end of the file is encountered. A typical schema for processfile is:

```
processfile :-
  read( Term),                  % Assuming Term not a variable
  process( Term).

process( end_of_file) :- !.     % All done

process( Term) :-
  treat( Term),                 % Process current item
  processfile.                  % Process rest of file
```

Here treat(Term) represents whatever is to be done with each term. An example would be a procedure to display on the terminal each term together with its consecutive number. Let us call this procedure showfile. It has to have an additional argument to count the terms read:

```
showfile( N) :-
  read( Term),
  show( Term, N).

show( end_of_file, _) :- !.

show( Term, N) :-
  write( N), tab( 2), write( Term), nl,
  N1 is N + 1,
  showfile( N1).
```

Exercises

6.1 Let f be a file of terms. Define a procedure

```
findterm( Term)
```

that displays on the terminal the first term in f that matches Term.

6.2 Let f be a file of terms. Write a procedure

```
findallterms( Term)
```

that displays on the terminal all the terms in f that match Term. Make sure that Term is not instantiated in the process (which could prevent its match with terms that occur later in the file).

6.3 Manipulating characters

A character is written on the current output stream with the goal

```
put( C)
```

where C is the ASCII code (a number between 0 and 127) of the character to be output. For example, the question

```
?- put( 65), put( 66), put( 67).
```

would cause the following output:

```
ABC
```

65 is the ASCII code of 'A', 66 of 'B', 67 of 'C'.

A single character can be read from the current input stream by the goal

```
get0( C)
```

This causes the current character to be read from the input stream, and the variable C becomes instantiated to the ASCII code of this character. A variation of the predicate get0 is get, which is used for reading non-blank characters. So the goal

```
get( C)
```

will cause the skipping over of all non-printable characters (blanks in particular) from the current input position in the input stream up to the first printable character. This character is then also read and C is instantiated to its ASCII code.

As an example of using predicates that transfer single characters let us define a procedure, squeeze, to do the following: read a sentence from the current input stream, and output the same sentence reformatted so that multiple blanks between words are replaced by single blanks. For simplicity we will assume that any input

sentence processed by squeeze ends with a full stop and that words are separated simply by one or more blanks, but no other character. An acceptable input is then:

```
The        robot tried     to pour wine out     of the       bottle.
```

The goal squeeze would output this in the form:

```
The robot tried to pour wine out of the bottle.
```

The squeeze procedure will have a similar structure to the procedures for processing files in the previous section. First it will read the first character, output this character, and then complete the processing depending on this character. There are three alternatives that correspond to the following cases: the character is either a full stop, a blank or a letter. The mutual exclusion of the three alternatives is achieved in the program by cuts:

```
squeeze :-
  get0( C),
  put( C),
  dorest( C).

dorest( 46) :- !.                    % 46 is ASCII for full stop, all done

dorest( 32) :- !,                    % 32 is ASCII for blank
  get( C),                           % Skip other blanks
  put( C),
  dorest( C).

dorest( Letter) :-
  squeeze.
```

Exercise

6.3 Generalize the squeeze procedure to handle commas as well. All blanks immediately preceding a comma are to be removed, and we want to have one blank after each comma.

6.4 Constructing and decomposing atoms

It is often desirable to have information, read as a sequence of characters, represented in the program as an atom. There is a built-in predicate, name, which can be used to this end. name relates atoms and their ASCII encodings. Thus,

```
name( A, L)
```

is true if L is the list of ASCII codes of the characters in A. For example,

```
name( zx232, [122,120,50,51,50] )
```

is true. There are two typical uses of name:

(1) given an atom, break it down into single characters;

(2) given a list of characters, combine them into an atom.

An example of the first kind of application would be a program that deals with orders, taxis and drivers. These would be, in the program, represented by atoms such as:

```
order1, order2, driver1, driver2, taxia1, taxilux
```

The following predicate:

```
taxi( X)
```

tests whether an atom X represents a taxi:

```
taxi( X) :-
  name( X, Xlist),
  name( taxi, Tlist),
  conc( Tlist, _, Xlist).            % Is word 'taxi' prefix of X?
```

Predicates order and driver can be defined analogously.

The next example illustrates the use of combining characters into atoms. We will define a predicate:

```
getsentence( Wordlist)
```

that reads a free-form natural language sentence and instantiates Wordlist to some internal representation of the sentence. A natural choice for the internal representation, which would enable further processing of the sentence, is this: each word of the input sentence is represented as a Prolog atom; the whole sentence is represented as a list of atoms. For example, if the current input stream is:

Mary was pleased to see the robot fail.

then the goal getsentence(Sentence) will cause the instantiation:

```
Sentence = [ 'Mary', was, pleased, to, see, the, robot, fail]
```

For simplicity, we will assume that each sentence terminates with a full stop and that there are no punctuation symbols within the sentence.

The program is shown in Figure 6.2. The procedure getsentence first reads the current input character, Char, and then supplies this character to the procedure getrest to complete the job. getrest has to react properly according to three cases:

(1) Char is the full stop: then everything has been read.

(2) Char is the blank: ignore it, getsentence from rest of input.


```
/*
  Procedure getsentence reads in a sentence and combines the words into a list of atoms.
  For example
    getsentence( Wordlist)
  produces
    Wordlist = [ 'Mary', was, pleased, to, see, the, robot, fail]
  if the input sentence is:
    Mary was pleased to see the robot fail.
*/

getsentence( Wordlist) :-
  get0( Char),
  getrest( Char, Wordlist).

getrest( 46, [] ) :- !.                       % End of sentence: 46 = ASCII for '.'

getrest( 32, Wordlist) :- !,                  % 32 = ASCII for blank
  getsentence( Wordlist).                     % Skip the blank

getrest( Letter, [Word | Wordlist] ) :-
  getletters( Letter, Letters, Nextchar),     % Read letters of current word
  name( Word, Letters),
  getrest( Nextchar, Wordlist).

getletters( 46, [], 46) :- !.                 % End of word: 46 = full stop

getletters( 32, [], 32) :- !.                 % End of word: 32 = blank

getletters( Let, [Let | Letters], Nextchar) :-
  get0( Char),
  getletters( Char, Letters, Nextchar).
```

Figure 6.2 A procedure to transform a sentence into a list of atoms.

(3) Char is a letter: first read the word, Word, which begins with Char, and then use getsentence to read the rest of the sentence, producing Wordlist. The cumulative result is the list [Word | Wordlist].

The procedure that reads the characters of one word is:

```
getletters( Letter, Letters, Nextchar)
```

The three arguments are:

(1) Letter is the current letter (already read) of the word being read.

(2) Letters is the list of letters (starting with Letter) up to the end of the word.

(3) Nextchar is the input character that immediately follows the word read. Nextchar must be a non-letter character.

We conclude this example with a comment on the possible use of the getsentence procedure. It can be used in a program to process text in natural language. Sentences represented as lists of words are in a form that is suitable for further processing in Prolog. A simple example is to look for certain keywords in input sentences. A much more difficult task would be to understand the sentence; that is, to extract from the sentence its meaning, represented in some chosen formalism. This is an important research area of Artificial Intelligence, and is introduced in Chapter 21.

Exercises

6.4 Define the relation

```
starts( Atom, Character)
```

to check whether Atom starts with Character.

6.5 Define the procedure plural that will convert nouns into their plural form. For example:

```
?- plural( table, X).
X = tables
```

6.6 Write the procedure

```
search( KeyWord, Sentence)
```

that will, each time it is called, find a sentence in the current input file that contains the given KeyWord. Sentence should be in its original form, represented as a sequence of characters or as an atom (procedure getsentence of this section can be accordingly modified).

6.5 Reading programs

We can communicate our programs to the Prolog system by means of built-in predicates that *consult* or *compile* files with programs. The details of 'consulting' and compiling files depend on the implementation of Prolog. Here we look at some basic facilities that are available in many Prologs.

We tell Prolog to read a program from a file F with a goal of the form consult(F), for example:

```
?- consult( program3).
```

Depending on the implementation, the file name program3 will possibly have to have an extension indicating that it is a Prolog program file. The effect of this goal will be that all the clauses in file program3 are read and loaded into the memory. So they will be used by Prolog when answering further questions from the user. Another file may be 'consulted' at some later time during the same session. Basically, the effect is again that the clauses from this new file are added into the memory. However, details depend on the implementation and other circumstances. If the new file contains clauses about a procedure defined in the previously consulted file, then the new clauses may be simply added at the end of the current set of clauses, or the previous definition of this procedure may be entirely replaced by the new one.

Several files may be consulted by the same consult goal, for example:

```
?- consult( [ program3, program4, queens]).
```

Such a question can also be written more simply as:

```
?- [ program3, program4, queens].
```

Consulted programs are used by a Prolog *interpreter*. If a Prolog implementation also features a *compiler*, then programs can be loaded in a compiled form. This enables more efficient execution with a typical speed-up factor of 5 or 10 between the interpreted and compiled code. Programs are loaded into memory in the compiled form by the built-in predicate compile, for example:

```
?- compile( program3).
```

or

```
?- compile( [ program4, queens, program6]).
```

Compiled programs are more efficiently executed, but interpreted programs are easier to debug because they can be inspected and traced by Prolog's debugging facilities. Therefore an interpreter is typically used in the program development phase, and a compiler is used with the final program.

It should be noted, again, that the details of consulting and compiling files depend on the implementation of Prolog. Usually a Prolog implementation also allows the user to enter and edit the program interactively.

Summary

- Input and output (other than that associated with querying the program) is done using built-in procedures. This chapter introduced a simple and practical repertoire of such procedures that can be found in many Prolog implementations.
- This repertoire assumes that files are sequential. There is the *current input stream* and the *current output stream*. The user terminal is treated as a file called user.

- Switching between streams is done by:

see(File)	File becomes the current input stream
tell(File)	File becomes the current output stream
seen	close the current input stream
told	close the current output stream

- Files are read and written in two ways:

  as sequences of characters
  as sequences of terms

  Built-in procedures for reading and writing characters and terms are:

read(Term)	input next term
write(Term)	output Term
put(CharCode)	output character with the given ASCII code
get0(CharCode)	input next character
get(CharCode)	input next 'printable' character

- Two procedures help formatting:

nl	output new line
tab(N)	output N blanks

- The procedure name(Atom, CodeList) decomposes and constructs atoms. CodeList is the list of ASCII codes of the characters in Atom.
- Many Prolog implementations provide additional facilities to handle non-sequential files, windows, provide graphics primitives, input information from the mouse, etc.

Reference to Prolog standard

For some of the predicates mentioned in this chapter, ISO standard for Prolog (Deransart *et al.* 1996) recommends different names from those used in most Prolog implementations. However, the predicates are conceptually the same, so compatibility is only a matter of renaming. The concerned predicates in this chapter are: see(Filename), tell(Filename), get(Code), put(Code), name(Atom,CodeList). The corresponding predicate names in the standard are: set_input(Filename), set_output(Filename), get_code(Code), put_code(Code), atom_codes(Atom,CodeList).

Deransart, P., Ed-Bdali, A. and Ceroni, L. (1996) *Prolog: The Standard.* Berlin: Springer-Verlag.

chapter 7

More Built-in Predicates

In this chapter we will examine some more built-in predicates for advanced Prolog programming. These features enable the programming of operations that are not possible using only the features introduced so far. One set of such predicates manipulate terms: testing whether some variable has been instantiated to an integer, taking terms apart, constructing new terms, etc. Another useful set of procedures manipulates the 'database': they add new clauses to the program or remove existing ones.

The built-in predicates largely depend on the implementation of Prolog. However, the predicates discussed in this chapter are provided by many Prolog implementations. Various implementations may provide additional features.

7.1 Testing the type of terms

7.1.1 Predicates *var, nonvar, atom, integer, float, number, atomic, compound*

Terms may be of different types: variable, integer, atom, etc. If a term is a variable then it can be, at some point during the execution of the program, instantiated or

uninstantiated. Further, if it is instantiated, its value can be an atom, a structure, etc. It is sometimes useful to know what the type of this value is. For example, we may want to add the values of two variables, X and Y, by:

```
Z is X + Y
```

Before this goal is executed, X and Y have to be instantiated to numbers. If we are not sure that X and Y will indeed be instantiated to numbers at this point then we should check this in the program before arithmetic is done.

To this end we can use the built-in predicate **number**. number(X) is true if X is a number or if it is a variable whose value is a number. We say that X must 'currently stand for' a number. The goal of adding X and Y can then be protected by the following test on X and Y:

```
..., number( X), number( Y), Z is X + Y, ...
```

If X and Y are not both numbers then no arithmetic will be attempted. So the number goals 'guard' the goal Z is X + Y before meaningless execution.

Built-in predicates of this sort are: **var**, **nonvar**, **atom**, **integer**, **float**, **number**, **atomic**, **compound**. Their meaning is as follows:

var(X)	succeeds if X is currently an uninstantiated variable
nonvar(X)	succeeds if X is not a variable, or X is an already instantiated variable
atom(X)	is true if X currently stands for an atom
integer(X)	is true if X currently stands for an integer
float(X)	is true if X currently stands for a real number
number(X)	is true if X currently stands for a number
atomic(X)	is true if X currently stands for a number or an atom
compound(X)	is true if X currently stands for a compound term (a structure)

The following example questions to Prolog illustrate the use of these built-in predicates:

```
?- var( Z), Z = 2.

Z = 2

?- Z = 2, var( Z).

no

?- integer( Z), Z = 2.

no

?- Z = 2, integer( Z), nonvar( Z).

Z = 2

?- atom( 3.14).

no
```



```
?- atomic( 3.14).

yes

?- atom( ==> ).

yes

?- atom( p(1) ).

no

?- compound ( 2 + X)

yes
```

We will illustrate the need for **atom** by an example. We would like to count how many times a given atom occurs in a given list of objects. To this purpose we will define a procedure:

```
count( A, L, N)
```

where A is the atom, L is the list and N is the number of occurrences. The first attempt to define **count** could be:

```
count( _, [], 0).

count( A, [A | L], N) :- !,
  count( A, L, N1),                % N1 = number of occurrences in tail
  N is N1 + 1.

count( A, [_ | L], N) :-
  count( A, L, N).
```

Now let us try to use this procedure on some examples:

```
?- count( a, [a,b,a,a], N).

N = 3

?- count( a, [a,b,X,Y], Na).

Na = 3

...

?- count( b, [a,b,X,Y], Nb).

Nb = 3

...

?- L = [a, b, X, Y], count( a, L, Na), count( b, L, Nb).

Na = 3
Nb = 1
X = a
Y = a
...
```

In the last example, X and Y both became instantiated to a and therefore we only got Nb = 1; but this is not what we had in mind. We are interested in the number of real occurrences of the given *atom*, and not in the number of terms that *match* this atom. According to this more precise definition of the count relation we have to check whether the head of the list is an atom. The modified program is as follows:

```
count( _, [ ], 0).

count( A, [B | L], N) :-
  atom( B), A = B, !,          % B is atom A?
  count( A, L, N1),            % Count in tail
  N is N1 + 1
  ;
  count( A, L, N).             % Otherwise just count the tail
```

The following, more complex programming exercise in solving cryptarithmetic puzzles makes use of the nonvar predicate.

7.1.2 A cryptarithmetic puzzle using *nonvar*

A popular example of a cryptarithmetic puzzle is

```
  D O N A L D
+ G E R A L D
  -----------
  R O B E R T
```

The problem here is to assign decimal digits to the letters D, O, N, etc., so that the above sum is valid. All letters have to be assigned different digits, otherwise trivial solutions are possible – for example, all letters equal zero.

We will define a relation

```
sum( N1, N2, N)
```

where N1, N2 and N represent the three numbers of a given cryptarithmetic puzzle. The goal sum(N1, N2, N) is true if there is an assignment of digits to letters such that N1 + N2 = N.

The first step toward a solution is to decide how to represent the numbers N1, N2 and N in the program. One way of doing this is to represent each number as a list of decimal digits. For example, the number 225 would be represented by the list [2,2,5]. As these digits are not known in advance, an uninstantiated variable will stand for each digit. Using this representation, the problem can be depicted as:

```
  [ D, O, N, A, L, D ]
+ [ G, E, R, A, L, D ]
= [ R, O, B, E, R, T ]
```


$Number1 = [D_{11}, D_{12}, \ldots, D_{1i}, \ldots]$
$Number2 = [D_{21}, D_{22}, \ldots, D_{2i}, \ldots]$
$Number3 = [D_{31}, D_{32}, \ldots, D_{3i}, \ldots]$

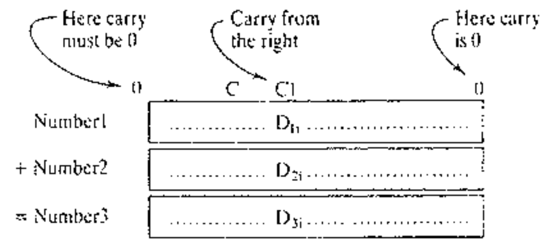


Figure 7.1 Digit by digit summation. The relations at the indicated ith digit position are: $D_{3i} = (C1 + D_{1i} + D_{2i}) \bmod 10$; $C = (C1 + D_{1i} + D_{2i})$ div 10.

The task is to find such an instantiation of the variables D, O, N, etc., for which the sum is valid. When the sum relation has been programmed, the puzzle can be stated to Prolog by the question:

```
?- sum( [D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T] ).
```

To define the sum relation on lists of digits, we have to implement the actual rules for doing summation in the decimal number system. The summation is done digit by digit, starting with the right-most digits, continuing toward the left, always taking into account the carry digit from the right. It is also necessary to maintain a set of available digits; that is, digits that have not yet been used for instantiating variables already encountered. So, in general, besides the three numbers N1, N2 and N, some additional information is involved, as illustrated in Figure 7.1:

- carry digit before the summation of the numbers;
- carry digit after the summation;
- set of digits available before the summation;
- remaining digits, not used in the summation.

To formulate the sum relation we will use, once again, the principle of generalization of the problem: we will introduce an auxiliary, more general relation, sum1. sum1 has some extra arguments, which correspond to the foregoing additional information:

```
sum1( N1, N2, N, C1, C, Digits1, Digits)
```

N1, N2 and N are our three numbers, as in the sum relation, C1 is carry from the right (before summation of N1 and N2), and C is carry to the left (after the summation). The following example illustrates:

```
?- sum1( [H,E], [6,E], [U,S], 1, 1, [1,3,4,7,8,9], Digits).

H = 8
E = 3
S = 7
U = 4
Digits = [1,9]
```

This corresponds to the following summation:

```
1 ←   ← 1
   8 3
   6 3
   ---
   4 7
```

As Figure 7.1 shows, C1 and C have to be 0 if N1, N2 and N are to satisfy the sum relation. Digits1 is the list of available digits for instantiating the variables in N1, N2 and N; Digits is the list of digits that were not used in the instantiation of these variables. Since we allow the use of any decimal digit in satisfying the sum relation, the definition of sum in terms of sum1 is as follows:

```
sum( N1, N2, N) :-
  sum1( N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
```

The burden of the problem has now shifted to the sum1 relation. This relation is, however, general enough that it can be defined recursively. We will assume, without loss of generality, that the three lists representing the three numbers are of equal length. Our example problem, of course, satisfies this constraint; if not, a 'shorter' number can be prefixed by zeros.

The definition of sum1 can be divided into two cases:

(1) The three numbers are represented by empty lists. Then:

```
sum1( [], [], [], C, C, Digs, Digs).
```

(2) All three numbers have some left-most digit and the remaining digits on their right. So they are of the form:

```
[D1 | N1],     [D2 | N2],     [D | N]
```

In this case two conditions must be satisfied:

(a) The three numbers N1, N2 and N have to satisfy the sum1 relation, giving some carry digit, C2, to the left, and leaving some unused subset of decimal digits, Digs2.

(b) The left-most digits D1, D2 and D, and the carry digit C2 have to satisfy the relation indicated in Figure 7.1: C2, D1 and D2 are added giving D and a carry to the left. This condition will be formulated in our program as a relation digitsum.

Translating this case into Prolog we have:

```
sum1( [D1 | N1], [D2 | N2], [D | N], C1, C, Digs1, Digs) :-
  sum1( N1, N2, N, C1, C2, Digs1, Digs2),
  digitsum( D1, D2, C2, D, C, Digs2, Digs).
```

It only remains to define the digitsum relation in Prolog. There is one subtle detail that involves the use of the metalogical predicate nonvar. D1, D2 and D have to be decimal digits. If any of them is not yet instantiated then it has to become instantiated to one of the digits in the list Digs2. This digit has to be deleted from the set of available digits. If D1, D2 or D is already instantiated then, of course, none

```
% Solving cryptarithmetic puzzles

sum( N1, N2, N) :-                          % Numbers represented as lists of digits
  sum1( N1, N2, N,
        0, 0,                               % Carries from right and to left both 0
        [0,1,2,3,4,5,6,7,8,9], _).          % All digits available

sum1( [], [], [], C, C, Digits, Digits).

sum1( [D1 | N1], [D2 | N2], [D | N], C1, C, Digs1, Digs) :-
  sum1( N1, N2, N, C1, C2, Digs1, Digs2),
  digitsum( D1, D2, C2, D, C, Digs2, Digs).

digitsum( D1, D2, C1, D, C, Digs1, Digs) :-
  del_var( D1, Digs1, Digs2),               % Select an available digit for D1
  del_var( D2, Digs2, Digs3),               % Select an available digit for D2
  del_var( D, Digs3, Digs),                 % Select an available digit for D
  S is D1 + D2 + C1,
  D is S mod 10,                            % Remainder
  C is S // 10.                             % Integer division

del_var( A, L, L) :-
  nonvar( A), !.                            % A already instantiated

del_var( A, [A | L], L).                    % Delete the head

del_var( A, [B | L], [B | L1] ) :-
  del_var( A, L, L1).                       % Delete from tail

% Some puzzles

puzzle1( [D,O,N,A,L,D],
         [G,E,R,A,L,D],
         [R,O,B,E,R,T] ).

puzzle2( [0,S,E,N,D],
         [0,M,O,R,E],
         [M,O,N,E,Y] ).
```

Figure 7.2 A program for cryptarithmetic puzzles.

of the available digits will be spent. This is realized in the program as a non-deterministic deletion of an item from a list. If this item is non-variable then nothing is deleted (no instantiation occurs). This is programmed as:

```
del_var( Item, List, List) :-
  nonvar( Item), !.                  % Item already instantiated

del_var( Item, [Item | List], List).  % Delete the head

del_var( Item, [A | List], [A | List1] ) :-
  del_var( Item, List, List1).        % Delete Item from tail
```

A complete program for cryptarithmetic puzzles is shown in Figure 7.2. The program also includes the definition of two puzzles. The question to Prolog about DONALD, GERALD and ROBERT, using this program, would be:

```
?- puzzle1( N1, N2, N), sum( N1, N2, N).
```

Sometimes this puzzle is made easier by providing part of the solution as an additional constraint that D be equal 5. The puzzle in this form could be communicated to Prolog using sum1:

```
?- sum1( [5,O,N,A,L,5],
         [G,E,R,A,L,5],
         [R,O,B,E,R,T],
         0, 0, [0,1,2,3,4,6,7,8,9], _).
```

It is interesting that in both cases there is only one solution. That is, there is only one way of assigning digits to letters.

Exercises

7.1 Write a procedure simplify to symbolically simplify summation expressions with numbers and symbols (lower-case letters). Let the procedure rearrange the expressions so that all the symbols precede numbers. These are examples of its use:

```
?- simplify( 1 + 1 + a, E).
E = a + 2
?- simplify( 1 + a + 4 + 2 + b + c, E).
E = a + b + c + 7
?- simplify( 3 + x + x, E).
E = 2*x + 3
```

7.2 Define the procedure:

```
add_to_tail( Item, List)
```


to store a new element into a list. Assume that all of the elements that can be stored are non-variables. List contains all the stored elements followed by a tail that is not instantiated and can thus accommodate new elements. For example, let the existing elements stored be a, b and c. Then

```
List = [a, b, c | Tail]
```

where Tail is a variable. The goal

```
add_to_tail( d, List)
```

will cause the instantiation

```
Tail = [d | NewTail]   and   List = [a, b, c, d | NewTail]
```

Thus the structure can, in effect, grow by accepting new items. Define also the corresponding membership relation.

7.2 Constructing and decomposing terms: =.., *functor, arg, name*

There are three built-in predicates for decomposing terms and constructing new terms: functor, arg and '=..'. We will first look at =.., which is written as an infix operator and reads as 'univ'. The goal

```
Term =.. L
```

is true if L is a list that contains the principal functor of Term, followed by its arguments. The following examples illustrate:

```
?- f( a, b) =.. L.
L = [f, a, b]
?- T =.. [rectangle, 3, 5].
T = rectangle( 3, 5)
?- Z =.. [p, X, f(X,Y)].
Z = p( X, f(X,Y) )
```

Why would we want to decompose a term into its components – its functor and its arguments? Why construct a new term from a given functor and arguments? The following example illustrates the need for this.

Let us consider a program that manipulates geometric figures. Figures are squares, rectangles, triangles, circles, etc. They can, in the program, be represented as terms such that the functor indicates the type of figure, and the arguments specify the size of the figure, as follows:

```
square( Side)
triangle( Side1, Side2, Side3)
circle( R)
```

One operation on such figures can be enlargement. We can implement this as a three-argument relation

```
enlarge( Fig, Factor, Fig1)
```

where Fig and Fig1 are geometric figures of the same type (same functor), and the parameters of Fig1 are those of Fig multiplicatively enlarged by Factor. For simplicity, we will assume that all the parameters of Fig are already known; that is, instantiated to numbers, and so is Factor. One way of programming the enlarge relation is:

```
enlarge( square(A), F, square(A1) ) :-
  A1 is F*A.

enlarge( circle(R), F, circle(R1) ) :-
  R1 is F*R1.

enlarge( rectangle(A,B), F, rectangle(A1,B1) ) :-
  A1 is F*A, B1 is F*B.

...
```

This works, but it is awkward when there are many different figure types. We have to foresee all types that may possibly occur. Thus, we need an extra clause for each type although each clause says essentially the same thing: take the parameters of the original figure, multiply all the parameters by the factor, and make a figure of the same type with new parameters.

One (unsuccessful) attempt to handle, at least, all one-parameter figures with one clause could be:

```
enlarge( Type(Par), F, Type(Par1) ) :-
  Par1 is F*Par.
```

However, this is normally not allowed in Prolog because the functor has to be an atom; so the variable Type would not be accepted syntactically as a functor. The correct method is to use the predicate '=..'. Then the enlarge procedure can be stated completely generally, for any type of object, as follows:

```
enlarge( Fig, F, Fig1) :-
  Fig =.. [Type | Parameters],
  multiplylist( Parameters, F, Parameters1),
  Fig1 =.. [Type | Parameters1].

multiplylist( [], _, [] ).

multiplylist( [X | L], F, [X1 | L1] ) :-
  X1 is F*X, multiplylist( L, F, L1).
```

Our next example of using the '=..' predicate comes from symbolic manipulation of formulas where a frequent operation is to substitute some subexpression by another expression. We will define the relation

```
substitute( Subterm, Term, Subterm1, Term1)
```


as follows: if all occurrences of Subterm in Term are substituted by Subterm1 then we get Term1. For example:

```
?- substitute( sin(x), 2*sin(x)*f(sin(x)), t, F).

F = 2*t*f(t)
```

By 'occurrence' of Subterm in Term we will mean something in Term that *matches* Subterm. We will look for occurrences from top to bottom. So the goal

```
?- substitute( a+b, f( a, A+B), v, F).
```

will produce

```
F = f( a, v)                  F = f( a, v+v)
A = a          and not        A = a+b
B = b                         B = a+b
```

In defining the substitute relation we have to consider the following decisions depending on the case:

If Subterm = Term then Term1 = Subterm1;
otherwise if Term is 'atomic' (not a structure)
then Term1 = Term (nothing to be substituted),
otherwise the substitution is to be carried out on the arguments of Term.

These rules can be converted into a Prolog program, shown in Figure 7.3.

```
% substitute( Subterm, Term, Subterm1, Term1):
%   if all occurrences of Subterm in Term are substituted with Subterm1 then we get Term1.

% Case 1: Substitute whole term

substitute( Term, Term, Term1, Term1) :- !.

% Case 2: Nothing to substitute if Term atomic

substitute( _, Term, _, Term) :-
  atomic( Term), !.

% Case 3: Do substitution on arguments

substitute( Sub, Term, Sub1, Term1) :-
  Term =.. [F | Args],                          % Get arguments
  substlist( Sub, Args, Sub1, Args1),           % Perform substitution on them
  Term1 =.. [F | Args1].

substlist( _, [], _, [] ).

substlist( Sub, [Term | Terms], Sub1, [Term1 | Terms1] ) :-
  substitute( Sub, Term, Sub1, Term1),
  substlist( Sub, Terms, Sub1, Terms1).
```

Figure 7.3 A procedure for substituting a subterm of a term by another subterm.

Terms that are constructed by the '=..' predicate can, of course, be also used as goals. The advantage of this is that the program itself can, during execution, generate and execute goals of forms that were not necessarily foreseen at the time of writing the program. A sequence of goals illustrating this effect would be something like the following:

```
obtain( Functor),
compute( Arglist),
Goal =.. [Functor | Arglist],
Goal
```

Here, obtain and compute are some user-defined procedures for getting the components of the goal to be constructed. The goal is then constructed by '=..', and invoked for execution by simply stating its name, Goal.

Some implementations of Prolog may require that all the goals, as they appear in the program, are *syntactically* either atoms or structures with an atom as the principal functor. Thus a variable, regardless of its eventual instantiation, in such a case may not be syntactically acceptable as a goal. This problem is circumvented by another built-in predicate, call, whose argument is the goal to be executed. Accordingly, the above example would be rewritten as:

```
...
Goal =.. [Functor | Arglist],
call( Goal)
```

Sometimes we may want to extract from a term just its principal functor or one of its arguments. We can, of course, use the '=..' relation. But it can be neater, and also more efficient, to use one of the other two built-in procedures for manipulating terms: functor and arg. Their meaning is as follows: a goal

```
functor( Term, F, N)
```

is true if F is the principal functor of Term and N is the arity of F. A goal

```
arg( N, Term, A)
```

is true if A is the Nth argument in Term, assuming that arguments are numbered from left to right starting with 1. The following examples illustrate:

```
?- functor( t( f(X), X, t), Fun, Arity).

Fun = t
Arity = 3

?- arg( 2, f( X, t(a), t(b)), Y).

Y = t(a)

?- functor( D, date, 3),
   arg( 1, D, 29),
   arg( 2, D, june),
   arg( 3, D, 1982).

D = date( 29, june, 1982)
```


The last example shows a special application of the functor predicate. The goal functor(D, date, 3) generates a 'general' term whose principal functor is date with three arguments. The term is general in that the three arguments are uninstantiated variables whose names are generated by Prolog. For example:

```
D = date( _5, _6, _7)
```

These three variables are then instantiated in the example above by the three arg goals.

Related to this set of built-in predicates is the predicate name for constructing/decomposing atoms, introduced in Chapter 6. We will repeat its meaning here for completeness.

```
name( A, L)
```

is true if L is the list of ASCII codes of the characters in atom A.

Exercises

7.3 Define the predicate ground(Term) so that it is true if Term does not contain any uninstantiated variables.

7.4 The substitute procedure of this section only produces the 'outer-most' substitution when there are alternatives. Modify the procedure so that all possible alternative substitutions are produced through backtracking. For example:

```
?- substitute( a+b, f(A+B), new, NewTerm).

A = a
B = b
NewTerm = f( new);

A = a+b
B = a+b
NewTerm = f( new+new)
```

Our original version only finds the first answer.

7.5 Define the relation

```
subsumes( Term1, Term2)
```

so that Term1 is more general than Term2. For example:

```
?- subsumes( X, c).

yes

?- subsumes( g(X), g(t(Y)) ).

yes

?- subsumes( f(X,X), f(a,b) ).

no
```

7.3 Various kinds of equality and comparison

When do we consider two terms to be equal? Until now we have introduced three kinds of equality in Prolog. The first was based on matching, written as:

```
X = Y
```

This is true if X and Y match. Another type of equality was written as:

```
X is E
```

This is true if X matches the value of the arithmetic expression E. We also had:

```
E1 =:= E2
```

This is true if the values of the arithmetic expressions E1 and E2 are equal. In contrast, when the values of two arithmetic expressions are not equal, we write:

```
E1 =\= E2
```

Sometimes we are interested in a stricter kind of equality: the *literal equality* of two terms. This kind of equality is implemented as another built-in predicate written as an infix operator '==':

```
T1 == T2
```

This is true if terms T1 and T2 are identical; that is, they have exactly the same structure and all the corresponding components are the same. In particular, the names of the variables also have to be the same. The complementary relation is 'not identical', written as:

```
T1 \== T2
```

Here are some examples:

```
?- f( a, b) == f( a, b).
yes
?- f( a, b) == f( a, X).
no
?- f( a, X) == f( a, Y).
no
?- X \== Y.
yes
?- t( X, f(a,Y) ) == t( X, f(a,Y) ).
yes
```

As an example, let us redefine the relation

count(Term, List, N)

from Section 7.1. This time let N be the number of literal occurrences of the term Term in a list List:

```
count( _, [], 0).
count( Term, [Head | L ], N) :-
  Term == Head, !,
  count( Term, L, N1),
  N is N1 + 1
  ;
  count( Term, L, N).
```

We have already seen predicates that compare terms arithmetically, for example X+2 < 5. Another set of built-in predicates compare terms alphabetically and thus define an ordering relation on terms. For example, the goal

```
X @< Y
```

is read: term X precedes term Y. The precedence between simple terms is determined by alphabetical or numerical ordering. The precedence between structures is determined by the precedence of their principal functors. If the principal functors are equal, then the precedence between the top-most, left-most functors in the subterms in X and Y decides. Examples are:

```
?- paul @< peter.
yes
?- f(2) @< f(3).
yes
?- g(2) @< f(3).
no
?- g(2) @>= f(3).
yes
?- f( a, g(b), c) @< f( a, h(a), a).
yes
```

All the built-in predicates in this family are @<, @=<, @>, @>= with their obvious meanings.

7.4 Database manipulation

According to the relational model of databases, a database is a specification of a set of relations. A Prolog program can be viewed as such a database: the specification of relations is partly explicit (facts) and partly implicit (rules). Some built-in predicates

make it possible to update this database during the execution of the program. This is done by adding (during execution) new clauses to the program or by deleting existing clauses. Predicates that serve these purposes are assert, asserta, assertz and retract.

A goal

```
assert( C)
```

always succeeds and, as its side effect, causes a clause C to be 'asserted' – that is, added to the database. A goal

```
retract( C)
```

does the opposite: it deletes a clause that matches C. The following conversation with Prolog illustrates:

```
?- crisis.
no
?- assert( crisis).
yes
?- crisis.
yes
?- retract( crisis).
yes
?- crisis.
no
```

Clauses thus asserted act exactly as part of the 'original' program. The following example shows the use of assert and retract as one method of handling changing situations. Let us assume that we have the following program about weather:

```
nice :-
  sunshine, not raining.

funny :-
  sunshine, raining.

disgusting :-
  raining, fog.

raining.

fog.
```

The following conversation with this program will gradually update the database:

```
?- nice.
no
```



```
?- disgusting.
yes
?- retract( fog).
yes
?- disgusting.
no
?- assert( sunshine).
yes
?- funny.
yes
?- retract( raining).
yes
?- nice.
yes
```

Clauses of any form can be asserted or retracted. However, depending on the implementation of Prolog, it may be required that predicates manipulated through assert/retract be declared as *dynamic*, using the directive dynamic(PredicateIndicator). Predicates that are only brought in by assert, and not by consult, are automatically assumed as dynamic.

The next example illustrates that retract is also non-deterministic: a whole set of clauses can, through backtracking, be removed by a single retract goal. Let us assume that we have the following facts in the 'consulted' program:

```
fast( ann).
slow( tom).
slow( pat).
```

We can add a rule to this program, as follows:

```
?- assert(
        ( faster(X,Y) :-
             fast(X), slow(Y) ) ).

yes

?- faster( A, B).

A = ann
B = tom

?- retract( slow(X) ).

X = tom;
X = pat;
no
```

```
?- faster( ann, _).
no
```

Notice that when a rule is asserted, the syntax requires that the rule (as an argument to assert) be enclosed in parentheses.

When asserting a clause, we may want to specify the position at which the new clause is inserted to the database. The predicates asserta and assertz enable us to control the position of insertion. The goal

```
asserta( C)
```

adds C at the beginning of the database. The goal

```
assertz( C)
```

adds C at the end of the database. We will assume that assert is equivalent to assertz, as usual in Prolog implementations. The following example illustrates these effects:

```
?- assert( p(b) ), assertz( p(c) ), assert( p(d) ), asserta( p(a) ).
yes

?- p( X).
X = a;
X = b;
X = c;
X = d
```

There is a relation between consult and assertz. Consulting a file can be defined in terms of assertz as follows: to consult a file, read each term (clause) in the file and assert it at the end of the database.

One useful application of asserta is to store already computed answers to questions. For example, let there be a predicate

```
solve( Problem, Solution)
```

defined in the program. We may now ask some question and request that the answer be remembered for future questions.

```
?- solve( problem1, Solution),
   asserta( solve( problem1, Solution) ).
```

If the first goal above succeeds then the answer (Solution) is stored and used, as any other clause, in answering further questions. The advantage of such a 'memoization' of answers is that a further question that matches the asserted fact will normally be answered much quicker than the first one. The result now will be simply retrieved as a fact, and not computed through a possibly time-consuming process. This technique of storing derived solutions is also called 'caching'.

An extension of this idea is to use asserting for generating all solutions in the form of a table of facts. For example, we can generate a table of products of all pairs of integers between 0 and 9 as follows: generate a pair of integers X and Y, compute Z is X*Y, assert the three numbers as one line of the product table, and then force the failure. The failure will cause, through backtracking, another pair of integers to be found and so another line tabulated, etc. The following procedure maketable implements this idea:

```
maketable :-
  L = [0,1,2,3,4,5,6,7,8,9],
  member( X, L),                % Choose first factor
  member( Y, L),                % Choose second factor
  Z is X*Y,
  assert( product(X,Y,Z) ),
  fail.
```

The question

```
?- maketable.
```

will, of course, not succeed, but it will, as a side effect, add the whole product table to the database. After that we can ask, for example, what pairs give the product 8:

```
?- product( A, B, 8).
A = 1
B = 8;

A = 2
B = 4;

...
```

A remark on the style of programming should be made at this stage. The foregoing examples illustrate some obviously useful applications of assert and retract. However, their use requires special care. Excessive and careless use of these facilities cannot be recommended as good programming style. By asserting and retracting we, in fact, modify the program. Therefore relations that hold at some point will not be true at some other time. At different times the same questions receive different answers. A lot of asserting and retracting may thus obscure the meaning of the program. The resulting behaviour of the program may become difficult to understand, difficult to explain and to trust.

Exercises

7.6 (a) Write a Prolog question to remove the whole product table from the database.

(b) Modify the question so that it only removes those entries where the product is 0.

7.7 Define the relation

```
copy_term( Term, Copy)
```

which will produce a copy of Term so that Copy is Term with all its variables renamed. This can be easily programmed by using asserta and retract. In some Prologs copy_term is provided as a built-in predicate.

7.5 Control facilities

So far we have covered most of the extra control facilities except repeat. For completeness the complete set is presented here.

- *cut*, written as '!', prevents backtracking. It was introduced in Chapter 5. A useful predicate is once(P) defined in terms of cut as:

```
once( P) :- P, !.
```

  once(P) produces one solution only. The cut, nested in once, does not prevent backtracking in other goals.
- fail is a goal that always fails.
- true is a goal that always succeeds.
- not(P) is negation as failure that behaves exactly as if defined as:

```
not( P) :- P, !, fail; true.
```

  Some problems with cut and not were discussed in detail in Chapter 5.
- call(P) invokes a goal P. It succeeds if P succeeds.
- repeat is a goal that always succeeds. Its special property is that it is non-deterministic; therefore, each time it is reached by backtracking it generates another alternative execution branch. repeat behaves as if defined by:

```
repeat.
repeat :- repeat.
```

  A typical way of using repeat is illustrated by the following procedure dosquares which reads a sequence of numbers and outputs their squares. The sequence is concluded with the atom stop, which serves as a signal for the procedure to terminate.

```
dosquares :-
  repeat,
  read( X),
  ( X = stop, !
   ;
   Y is X*X, write(Y),
   fail
  ).
```


7.6 *bagof*, *setof* and *findall*

We can generate, by backtracking, all the objects, one by one, that satisfy some goal. Each time a new solution is generated, the previous one disappears and is not accessible any more. However, sometimes we would prefer to have all the generated objects available together – for example, collected into a list. The built-in predicates bagof, setof and findall serve this purpose.

The goal

```
bagof( X, P, L)
```

will produce the list L of all the objects X such that a goal P is satisfied. Of course, this usually makes sense only if X and P have some common variables. For example, let us have these facts in the program:

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).
```

Then we can obtain the list of all the children of age 5 by the goal:

```
?- bagof( Child, age( Child, 5), List).

List = [ ann, tom]
```

If, in the above goal, we leave the age unspecified, then we get, through backtracking, three lists of children, corresponding to the three age values:

```
?- bagof( Child, age( Child, Age), List).

Age = 7
List = [ peter];

Age = 5
List = [ ann, tom];

Age = 8
List = [ pat];

no
```

We may prefer to have all of the children in one list regardless of their age. This can be achieved by explicitly stating in the call of bagof that we do not care about the value of Age as long as such a value exists. This is stated as:

```
?- bagof( Child, Age ^ age( Child, Age), List).

List = [ peter, ann, pat, tom]
```

Syntactically, '^' is a predefined infix operator of type xfy.

If there is no solution for P in the goal bagof(X, P, L), then the bagof goal simply fails. If the same object X is found repeatedly, then all of its occurrences will appear in L, which leads to duplicate items in L.

The predicate setof is similar to bagof. The goal

```
setof( X, P, L)
```

will again produce a list L of objects X that satisfy P. Only this time the list L will be ordered, and duplicate items, if there are any, will be eliminated. The ordering of the objects is according to built-in predicate @<, which defines the precedence among terms. For example:

```
?- setof( Child, Age ^ age( Child, Age), ChildList),
   setof( Age, Child ^ age( Child, Age), AgeList).

ChildList = [ ann, pat, peter, tom]
AgeList = [ 5, 7, 8]
```

There is no restriction on the kind of objects that are collected. So we can, for example, construct the list of children ordered by their age, by collecting pairs of the form Age/Child:

```
?- setof( Age/Child, age( Child, Age), List).

List = [ 5/ann, 5/tom, 7/peter, 8/pat]
```

Another predicate of this family, similar to bagof, is findall.

```
findall( X, P, L)
```

produces, again, a list of objects that satisfy P. The difference with respect to bagof is that *all* of the objects X are collected regardless of (possibly) different solutions for variables in P that are not shared with X. This difference is shown in the following example:

```
?- findall( Child, age( Child, Age), List).

List = [ peter, ann, pat, tom]
```

If there is no object X that satisfies P then findall will succeed with L = [ ].

If findall is not available as a built-in predicate in the implementation used then it can be easily programmed as follows. All solutions for P are generated by forced backtracking. Each solution is, when generated, immediately asserted into the database so that it is not lost when the next solution is found. After all the solutions have been generated and asserted, they have to be collected into a list and retracted from the database. This whole process can be imagined as all the solutions generated forming a queue. Each newly generated solution is, by assertion, added to the end of this queue. When the solutions are collected the queue dissolves. Note, in addition, that the end of this queue has to be marked, for example, by the atom 'bottom' (which, of course, should be different from any solution that is possibly expected). An implementation of findall along these lines is shown as Figure 7.4.


```
findall( X, Goal, Xlist) :-
  call( Goal),                      % Find a solution
  assertz( queue(X) ),              % Assert it
  fail;                             % Try to find more solutions
  assertz( queue(bottom) ),         % Mark end of solutions
  collect( Xlist).                  % Collect the solutions

collect( L) :-
  retract( queue(X) ), !,           % Retract next solution
  ( X == bottom, !, L = [ ]         % End of solutions?
    ;
    L = [X | Rest], collect( Rest) ).  % Otherwise collect the rest
```

Figure 7.4 An implementation of the findall relation.

Exercises

7.8 Use bagof to define the relation powerset(Set, Subsets) to compute the set of all subsets of a given set (all sets represented as lists).

7.9 Use bagof to define the relation

```
copy_term( Term, Copy)
```

such that Copy is Term with all its variables renamed.

Summary

- A Prolog implementation normally provides a set of built-in procedures to accomplish several useful operations that are not possible in pure Prolog. In this chapter, such a set of predicates available in many Prolog implementations was introduced.
- The type of a term can be tested by the following predicates:

var(X)	X is a (non-instantiated) variable
nonvar(X)	X is not a variable
atom(X)	X is an atom
integer(X)	X is an integer
float(X)	X is a real number
atomic(X)	X is either an atom or a number
compound(X)	X is a structure

- Terms can be constructed or decomposed:

```
Term =.. [ Functor | ArgumentList]
functor( Term, Functor, Arity)
arg( N, Term, Argument)
name( Atom, CharacterCodes)
```

- Terms can be compared:

X = Y	X and Y match
X == Y	X and Y are identical
X \== Y	X and Y are not identical
X =:= Y	X and Y are arithmetically equal
X =\= Y	X and Y are not arithmetically equal
X < Y	arithmetic value of X is less than Y (related: =<, >, >=)
X @< Y	term X precedes term Y (related: @=<, @>, @>=)

- A Prolog program can be viewed as a relational database that can be updated by the following procedures:

assert(Clause)	add Clause to the program
asserta(Clause)	add at the beginning
assertz(Clause)	add at the end
retract(Clause)	remove a clause that matches Clause

- All the objects that satisfy a given condition can be collected into a list by the predicates:

bagof(X, P, L)	L is the list of all X that satisfy condition P
setof(X, P, L)	L is the sorted list of all X that satisfy condition P
findall(X, P, L)	similar to bagof

- repeat is a control facility that generates an unlimited number of alternatives for backtracking.

chapter 8

Programming Style and Technique

In this chapter we will review some general principles of good programming and discuss the following questions in particular: How to think about Prolog programs? What are elements of good programming style in Prolog? How to debug Prolog programs? How to make Prolog programs more efficient?

8.1 General principles of good programming

What is a good program? Answering this question is not trivial as there are several criteria for judging how good a program is. Generally accepted criteria include the following:

- *Correctness* Above all, a good program should be correct. That is, it should do what it is supposed to do. This may seem a trivial, self-explanatory requirement. However, in the case of complex programs, correctness is often not attained. A common mistake when writing programs is to neglect this obvious criterion and pay more attention to other criteria, such as efficiency or external glamour of the program.
- *User-friendliness* A good program should be easy to use and interact with.
- *Efficiency* A good program should not needlessly waste computer time and memory space.

- *Readability* A good program should be easy to read and easy to understand. It should not be more complicated than necessary. Clever programming tricks that obscure the meaning of the program should be avoided. The general organization of the program and its layout help its readability.
- *Modifiability* A good program should be easy to modify and to extend. Transparency and modular organization of the program help modifiability.
- *Robustness* A good program should be robust. It should not crash immediately when the user enters some incorrect or unexpected data. The program should, in the case of such errors, stay 'alive' and behave reasonably (should report errors).
- *Documentation* A good program should be properly documented. The minimal documentation is the program's listing including sufficient program comments.

The importance of particular criteria depends on the problem and on the circumstances in which the program is written, and on the environment in which it is used. There is no doubt that correctness has the highest priority. The issues of readability, user-friendliness, modifiability, robustness and documentation are usually given, at least, as much priority as the issue of efficiency.

There are some general guidelines for practically achieving the above criteria. One important rule is to first *think* about the problem to be solved, and only then to start writing the actual code in the programming language used. Once we have developed a good understanding of the problem and the solution is well thought through, the actual coding will be fast and easy, and there is a good chance that we will soon get a correct program.

A common mistake is to start writing the code even before the full definition of the problem has been understood. A fundamental reason why early coding is bad practice is that the thinking about the problem and the ideas for a solution should be done in terms that are most relevant to the problem. These terms are usually far from the syntax of the programming language used, and they may include natural language statements and pictorial representation of ideas.

Such a formulation of the solution will have to be transformed into the programming language, but this transformation process may not be easy. A good approach is to use the principle of *stepwise refinement*. The initial formulation of the solution is referred to as the 'top-level solution', and the final program as the 'bottom-level solution'.

According to the principle of stepwise refinement, the final program is developed through a sequence of transformations, or 'refinements', of the solution. We start with the first, top-level solution and then proceed through a sequence of solutions; these are all equivalent, but each solution in the sequence is expressed in more detail. In each refinement step, concepts used in previous formulations are elaborated to greater detail and their representation gets closer to the programming language. It should be realized that refinement applies both to procedure definitions

and to data structures. In the initial stages we normally work with more abstract, bulky units of information whose structure is refined later.

Such a strategy of top-down stepwise refinement has the following advantages:

- it allows for formulation of rough solutions in terms that are most relevant to the problem;
- in terms of such powerful concepts, the solution should be succinct and simple, and therefore likely to be correct;
- each refinement step should be small enough so that it is intellectually manageable; if so, the transformation of a solution into a new, more detailed representation is likely to be correct, and so is the resulting solution at the next level of detail.

In the case of Prolog we may talk about the stepwise refinement of *relations*. If the problem suggests thinking in algorithmic terms, then we can also talk about refinement of *algorithms*, adopting the procedural point of view in Prolog.

In order to properly refine a solution at some level of detail, and to introduce useful concepts at the next lower level, we need ideas. Therefore programming is creative, especially so for beginners. With experience, programming gradually becomes less of an art and more of a craft. But, nevertheless, a major question is: How do we get ideas? Most ideas come from experience, from similar problems whose solutions we know. If we do not know a direct programming solution, another similar problem could be helpful. Another source of ideas is everyday life. For example, if the problem is to write a program to sort a list of items we may get an idea from considering the question: How would I myself sort a set of exam papers according to the alphabetical order of students?

General principles of good programming outlined in this section basically apply to Prolog as well. We will discuss some details with particular reference to Prolog in the following sections.

8.2 How to think about Prolog programs

One characteristic feature of Prolog is that it allows for both the procedural and declarative way of thinking about programs. The two approaches have been discussed in detail in Chapter 2, and illustrated by examples throughout the text. Which approach will be more efficient and practical depends on the problem. Declarative solutions are usually easier to develop, but may lead to an inefficient program.

During the process of developing a solution we have to find ideas for reducing problems to one or more easier subproblems. An important question is: How do we

find proper subproblems? There are several general principles that often work in Prolog programming. These will be discussed in the following sections.

8.2.1 Use of recursion

The principle here is to split the problem into cases belonging to two groups:

(1) trivial, or 'boundary' cases;

(2) 'general' cases where the solution is constructed from solutions of (simpler) versions of the original problem itself.

In Prolog we use this technique all the time. Let us look at one more example: processing a list of items so that each item is transformed by the same transformation rule. Let this procedure be

```
maplist( List, F, NewList)
```

where List is an original list, F is a transformation rule (a binary relation) and NewList is the list of all transformed items. The problem of transforming List can be split into two cases:

(1) Boundary case: List = [ ]

if List = [ ] then NewList = [ ], regardless of F

(2) General case: List = [X | Tail]

To transform a list of the form [X | Tail], do:
transform the item X by rule F obtaining NewX, and
transform the list Tail obtaining NewTail;
the whole transformed list is [NewX | NewTail].

In Prolog:

```
maplist( [ ], _, [ ] ).

maplist( [X | Tail], F, [NewX | NewTail] ) :-
  G =.. [F, X, NewX],
  call( G),
  maplist( Tail, F, NewTail).
```

Suppose we have a list of numbers and want to compute the list of their squares. maplist can be used for this as follows:

```
square( X, Y) :-
  Y is X*X.

?- maplist( [2, 6, 5], square, Squares).
Squares = [ 4, 36, 25]
```


One reason why recursion so naturally applies to defining relations in Prolog is that data objects themselves often have recursive structure. Lists and trees are such objects. A list is either empty (boundary case) or has a head and a tail that is itself a list (general case). A binary tree is either empty (boundary case) or it has a root and two subtrees that are themselves binary trees (general case). Therefore, to process a whole non-empty tree, we must do something with the root, and process the subtrees.

8.2.2 Generalization

It is often a good idea to generalize the original problem, so that the solution to the generalized problem can be formulated recursively. The original problem is then solved as a special case of its more general version. Generalization of a relation typically involves the introduction of one or more extra arguments. A major problem, which may require deeper insight into the problem, is how to find the right generalization.

As an example let us revisit the eight queens problem. The original problem was to place eight queens on the chessboard so that they do not attack each other. Let us call the corresponding relation:

```
eightqueens( Pos)
```

This is true if Pos is a position with eight non-attacking queens. A good idea in this case is to generalize the number of queens from eight to N. The number of queens now becomes the additional argument:

```
nqueens( Pos, N)
```

The advantage of this generalization is that there is an immediate recursive formulation of the nqueens relation:

(1) Boundary case: N = 0

To safely place zero queens is trivial.

(2) General case: N > 0

To safely place N queens on the board, satisfy the following:

- achieve a safe configuration of (N − 1) queens; and
- add the remaining queen so that she does not attack any other queen.

Once the generalized problem has been solved, the original problem is easy:

```
eightqueens( Pos) :- nqueens( Pos, 8).
```

8.2.3 Using pictures

When searching for ideas about a problem, it is often useful to introduce some graphical representation of the problem. A picture may help us to perceive some essential relations in the problem. Then we just have to describe what we *see* in the picture in the programming language.

The use of pictorial representations is often useful in problem solving in general; it seems, however, that it works with Prolog particularly well. The following arguments explain why:

(1) Prolog is particularly suitable for problems that involve objects and relations between objects. Often, such problems can be naturally illustrated by graphs in which nodes correspond to objects and arcs correspond to relations.

(2) Structured data objects in Prolog are naturally pictured as trees.

(3) The declarative meaning of Prolog facilitates the translation of pictorial representations into Prolog because, in principle, the order in which the picture is described does not matter. We just put what we see into the program in any order. (For practical reasons of the program's efficiency this order will possibly have to be polished later.)

8.3 Programming style

The purpose of conforming to some stylistic conventions is:

- to reduce the danger of programming errors; and
- to produce programs that are readable and easy to understand, easy to debug and to modify.

We will review here some ingredients of good programming style in Prolog: some general rules of good style, tabular organization of long procedures and commenting.

8.3.1 Some rules of good style

- Program clauses should be short. Their body should typically contain no more than a few goals.
- Procedures should be short because long procedures are hard to understand. However, long procedures are acceptable if they have some uniform structure (this will be discussed later in this section).
- Mnemonic names for procedures and variables should be used. Names should indicate the meaning of relations and the role of data objects.

- The layout of programs is important. Spacing, blank lines and indentation should be consistently used for the sake of readability. Clauses about the same procedure should be clustered together; there should be blank lines between clauses (unless, perhaps, there are numerous facts about the same relation); each goal can be placed on a separate line. Prolog programs sometimes resemble poems for the aesthetic appeal of ideas and form.
- Stylistic conventions of this kind may vary from program to program as they depend on the problem and personal taste. It is important, however, that the same conventions are used consistently throughout the whole program.
- The cut operator should be used with care. Cut should not be used if it can be easily avoided. It is better to use, where possible, 'green cuts' rather than 'red cuts'. As discussed in Chapter 5, a cut is called 'green' if it can be removed without altering the declarative meaning of the clause. The use of 'red cuts' should be restricted to clearly defined constructs such as **not** or the selection between alternatives. An example of the latter construct is:

  if Condition *then* Goal1 *else* Goal2

  This translates into Prolog, using cut, as:

```
Condition, !,          % Condition true?
Goal1                  % If yes then Goal1
;
Goal2                  % Otherwise Goal2
```

- The **not** procedure can also lead to surprising behaviour, as it is related to cut. We have to be well aware of how **not** is defined in Prolog. However, if there is a dilemma between **not** and cut, the former is perhaps better than some obscure construct with cut.
- Program modification by **assert** and **retract** can grossly degrade the transparency of the program's behaviour. In particular, the same program will answer the same question differently at different times. In such cases, if we want to reproduce the same behaviour we have to make sure that the whole previous state, which was modified by assertions and retractions, is completely restored.
- The use of a semicolon may obscure the meaning of a clause. The readability can sometimes be improved by splitting the clause containing the semicolon into more clauses; but this will, possibly, be at the expense of the length of the program and its efficiency.

To illustrate some points of this section consider the relation

```
merge( List1, List2, List3)
```

where List1 and List2 are ordered lists that merge into List3. For example:

```
merge( [2,4,7], [1,3,4,8], [1,2,3,4,4,7,8] )
```